

Draft Specification of Transactional Language Constructs for C++

Version: 1.1

February 3, 2011

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 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 2009-2012 by IBM Corporation, Intel Corporation and Oracle America, Inc. (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
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	5
2. Transaction statement	6
2.1 Memory model	7
3. Relaxed transactions	8
3.1 The <code>transaction_callable</code> function attribute	9
3.2 Nesting	10
3.3 Examples	10
4. Atomic transactions	10
4.1 Outer atomic transactions	11
4.2 The <code>transaction_safe</code> and the <code>transaction_unsafe</code> attributes	11
4.3 Examples	14
4.4 Nesting	16
4.5 Memory model	17
5. Transaction expressions	17
6. Function transaction blocks	18
7. Noexcept specification	19
8. Cancel statement	20
8.1 The <code>outer</code> attribute on cancel statements	21
8.2 The <code>transaction_may_cancel_outer</code> attribute	21
8.3 Examples	23
8.4 Memory model	23
9. Cancel-and-throw statement	24
9.1 The <code>outer</code> attribute on cancel-and-throw statements	25
9.2 Examples	25
10. Inheritance and compatibility rules for attributes	27
11. Class attributes	28
Appendix A. Grammar	29
Appendix B. Feature dependences	30
Appendix C. Extensions	33
Appendix D. Changes compared to version 1.0	35

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

1 indicate that the outermost atomic transaction should be cancelled (Sections 8.1 and 9.1). Such
2 cancel and cancel-and-throw statements can execute only within the dynamic extent of a
3 transaction statement with the `outer` attribute. The `transaction_may_cancel_outer`
4 attribute for functions and function pointers facilitates compile-time enforcement of this rule.
5

6 If an exception escapes from an atomic transaction statement without it being explicitly cancelled,
7 the atomic transaction takes effect. Programmers can guard against subtle bugs caused by
8 exceptions escaping a transaction statement unexpectedly by using `noexcept` specifications
9 (Section 7) to specify if exceptions are (or are not) expected to be thrown from within an atomic
10 transaction. A runtime error occurs, which leads to program termination, if an exception escapes
11 the scope of an atomic transaction that has a `noexcept` specification specifying no exceptions
12 may escape its scope.
13

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

19 2. Transaction statement

20 The `__transaction_relaxed` or the `__transaction_atomic` keyword followed by a
21 compound statement defines a *transaction statement*, that is, a statement that executes as a
22 transaction:
23

```
24 __transaction_relaxed compound-statement  
25 __transaction_atomic compound-statement  
26
```

27 In a data-race-free program (Section 2.1), all transactions appear to execute sequentially in some
28 total order. This means that transactions execute in isolation from other transactions; that is, the
29 individual operations of a transaction appear not to interleave with individual operations of
30 another transaction.
31

32 [Note: *Although transactions behave as if they execute in some serial order, an implementation*
33 *(i.e., compiler, runtime, and hardware) is free to execute transactions concurrently while providing*
34 *the illusion of serial ordering.*]
35

36 A transaction statement defined by the `__transaction_relaxed` keyword specifies a *relaxed*
37 *transaction* (Section 3). A transaction statement defined by the `__transaction_atomic`
38 keyword specifies an *atomic transaction* (Section 4). Relaxed transactions have no restrictions on
39 the kind of operations they may contain, but provide only basic isolation guarantee of all
40 transactions – they appear to execute sequentially with respect to other transactions (both
41 relaxed and atomic). Relaxed transactions may appear to interleave with non-transactional
42 operations of another thread. Atomic transactions provide a stronger isolation guarantee; that is,
43 they do not appear to interleave with any operations of other threads. Atomic transactions,
44 however, may contain only “safe” code (Section 4.2).
45

46 A `goto` or `switch` statement must not be used to transfer control into a transaction statement. A
47 `goto`, `break`, `return`, or `continue` statement may be used to transfer control out of a
48 transaction statement. When this happens, each variable declared in the transaction statement
49 will be destroyed in the context that directly contains its declaration.
50

51 The body of a transaction statement may throw an exception that is not handled inside its body
52 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.

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

4
5 The transaction order contributes to the “synchronizes with” relationship defined in the C++11
6 standard. In particular, each EndTransaction operation synchronizes with the next
7 StartTransaction operation in the transaction order executed by a different thread.

8
9 [Note: *The definition of the “synchronizes with” relation affects all other parts of the memory
10 model, including the definition of the “happens before” relationship, visibility rules that specify
11 what values can be seen by the reads, and the definition of data race freedom. Consequently,
12 including transactions in the “synchronizes with” relation is the only change to the memory model
13 that is necessary to account for transaction statements. With this extension, the C++11 memory
14 model fully describes the behavior of programs with transaction statements.*]

15
16 [Note: *A shared memory access can form a data race even if it is performed in a transaction
17 statement. In the following example, a write by thread T2 forms a data race with both read and
18 write to x by Thread T1 because it is not ordered with the operations of Thread T1 by the
19 “happens-before” relationship. To avoid a data race in this example, a programmer should
20 enclose the write to x in Thread T2 in a transaction statement.*

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

22]
23
24

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

32 3. Relaxed transactions

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

35
36 `__transaction_relaxed` *compound-statement*

37
38 A relaxed transaction is a compound statement that executes without observing changes made
39 by other transactions during its execution. Furthermore, other threads’ transactions do not
40 observe partial results of concurrently executing transactions. Programmers can think of a
41 relaxed transaction statement as a sequence of operations that do not interleave with the
42 operations of other transactions, which simplifies reasoning about the interaction of concurrently
43 executing transactions of different threads.

44
45 Relaxed transactions have no restrictions on the kind of operations that can be placed inside of
46 them and, thus allow any non-transactional code to be wrapped in a transaction. This makes
47 relaxed transactions flexible with regard to their usability, thereby allowing them to communicate
48 with other threads and the external world (e.g., via locks, C++11 atomic variables, volatile
49 variables or I/O) while still isolating them from other transactions. However, relaxed transactions

1 that contain such external world operations are not guaranteed isolation, even in data-race-free
2 programs. Other threads that communicate with a transaction can observe partial results of the
3 transaction, and the transaction can observe actions of other threads during its execution.

4
5 The following example illustrates a data-race-free program in which a relaxed transaction
6 synchronizes with another thread via access to a C++11 atomic variable: Note that accesses to
7 variable `x` in Thread 1 do not form data races with accesses to `x` in Thread 2 because operations
8 on C++11 atomic variables cannot create a data race:
9

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>

10
11 Relaxed transactions appear to interleave with non-transactional actions of other threads only
12 when they perform non-transactional forms of synchronization, such as operations on locks or
13 C++11 atomic variables. Transactions that do not execute such actions appear to execute
14 atomically, that is, as single indivisible operations.

15
16 Relaxed transactions may execute operations with side effects that the system cannot roll back.
17 We refer to such operations as *irrevocable* actions. For example, communicating partial results of
18 a relaxed transaction to either the external world via an I/O operation or to other threads via a
19 synchronization operation (such as a lock release or a write to a C++11 atomic variable) may
20 constitute an irrevocable action because the system may not be able to roll back the effects that
21 this communication had on the external world or other threads. For this reason, relaxed
22 transactions cannot be cancelled (Section 8). Irrevocable actions may limit the concurrency in an
23 implementation; for example, they may cause the implementation to not execute relaxed
24 transactions concurrently with other transactions.

25 **3.1 The `transaction_callable` function attribute**

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

36
37 The `transaction_callable` attribute specifies a property of a specific function, not its type. It
38 cannot be associated with pointers to functions, and may not be used in a `typedef` declaration.
39

40 A function declared with the `transaction_callable` attribute must not be re-declared without
41 that attribute. A function declared without the `transaction_callable` attribute must not be re-
42 declared with the `transaction_callable` attribute. See Section 10 for rules and restrictions
43 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;	
Thread T1	Thread T2
<pre> __transaction_relaxed { int tmp = x; x = y; y = tmp; } </pre>	<pre> int tmpX = 0, tmpY = 0; __transaction_relaxed { tmpX = x; tmpY = y; } assert(tmpX != tmpY); </pre>

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."	
Thread T1	Thread T2
<pre> __transaction_relaxed { std::cout << "Hello World."; } </pre>	<pre> __transaction_relaxed { std::cout << "Hello World."; } </pre>

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.

1
2 An atomic transaction executes in an all-or-nothing fashion: it can be explicitly cancelled so that
3 its operations have no effect (Section 8).

4
5 These properties make it easier to reason about the interaction of atomic transactions and the
6 actions of other threads when compared to other synchronization mechanisms such as mutual
7 exclusion.

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

17 **4.1 Outer atomic transactions**

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

```
19  
20 __transaction_atomic [[ outer ]] compound-statement
```

21
22 An outer atomic transaction is an atomic transaction that must not be nested lexically or
23 dynamically within another atomic transaction. Thus, an outer atomic transaction must not
24 appear within an atomic transaction or within the body of a function that might be called inside an
25 atomic transaction (see Section 8.2) for details about how this is enforced).

26
27 Outer atomic transactions enable the use of the `cancel-outer` statement (Section 8.1), which can
28 be executed only within the dynamic extent of an outer atomic transaction.

29 **4.2 The `transaction_safe` and the `transaction_unsafe`** 30 **attributes**

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

37
38 Function declarations (including virtual and template function declarations), declarations of
39 function pointers, and `typedef` declarations involving function pointers may specify
40 `transaction_safe` or `transaction_unsafe` attributes. A function declared with the
41 `transaction_safe` attribute has a transaction-safe type, and may be called within the dynamic
42 extent of an atomic transaction. The `transaction_unsafe` attribute specifies a transaction-
43 unsafe type. A transaction-safe type might also be specified by implicitly declaring a function *safe*,
44 as described further in this section.

45
46 We sometimes abbreviate the statement that a function has transaction-safe or transaction-
47 unsafe type by stating simply that the function is transaction-safe or transaction-unsafe,
48 respectively.

49
50 A function type must not be both transaction-safe and transaction-unsafe. That is, function
51 declarations, function pointer declarations, or `typedef` declarations for function pointer types

1 must not specify both the `transaction_safe` and the `transaction_unsafe` attributes. If
2 any declaration of such an entity specifies the `transaction_safe` attribute then every such
3 declaration (except a function definition, if it is not a virtual function) must specify the
4 `transaction_safe` attribute. A function declaration that specifies the
5 `transaction_callable` attribute may also specify the `transaction_safe` or the
6 `transaction_unsafe` attribute.

7
8 *[Note: A function declared in multiple compilation units must have the same type in all of these
9 compilation units. For example, a function that has a transaction-safe type in one compilation unit
10 must be declared to have such a type in all compilation units where it is declared.]*

11
12 Pointers to transaction-safe functions are implicitly convertible to pointers to the corresponding
13 transaction-unsafe functions. Such conversions are treated as identity conversions for purposes
14 of overload resolution, i.e., they have no effect on the ranking of conversion sequences. There is
15 no conversion from transaction-unsafe function pointers to transaction-safe function pointers.

16
17 The `transaction_safe` and `transaction_unsafe` attributes specify properties of the type of
18 the declared object, or of a type declared using `typedef`. Although such properties are ignored
19 for overload resolution, they are part of the type and propagated as such. For example:

```
20  
21 auto f = []()[[transaction_safe]] { g(); }
```

22
23 declares `f()` to be transaction-safe.

24
25 An atomic transaction or a body of a function declared with the `transaction_safe` attribute
26 must not contain calls to transaction-unsafe functions and other unsafe statements, defined
27 precisely below. This ensures that such statements are not executed within the dynamic extent of
28 an atomic transaction.

29
30 A statement is *unsafe* if any of the following applies:

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

39
40
41 *[Note: A relaxed transaction is unsafe because it may contain unsafe statements (Section 3). An
42 outer atomic transaction is unsafe because it cannot be nested within another atomic transaction.
43 A statement that contains an initialization of, assignment to, or a read from a volatile object is
44 unsafe because a value of a volatile object may be changed by means undetectable to an
45 implementation. The definition of the unsafe `asm` declaration is implementation-defined because
46 the meaning of the `asm` declaration is implementation-defined.]*

47
48 Although built-in operators are safe, they may be overloaded with user-defined operators, which
49 result in function calls. Thus, applications of these operators may be safe or unsafe, as
50 determined by the rules defined in this section. (For example, although the built-in `new` and
51 `delete` operators are safe, user-defined `new` and `delete` operators may be unsafe. Atomic
52 operations defined by the standard library are unsafe.)

53
54 A function definition *implicitly declares a function safe*, that is, declares its type to be transaction-
55 safe, if the function is not a virtual function, its body contains only safe statements, and neither

1 the definition nor any prior declaration of the function specifies any of the
2 `transaction_unsafe`, `transaction_safe`, or `transaction_may_cancel_outer`
3 (section 8.2) attributes. (If the definition or a prior declaration specifies the `transaction_safe`
4 attribute, the function is of transaction-safe type, but the definition does not *implicitly* declare the
5 function safe.) If the definition of a function implicitly declares it safe then no declaration of that
6 function may specify the `transaction_unsafe` attribute. Note that a recursive function that
7 directly calls itself is never implicitly declared safe. It may, however, explicitly specify a
8 `transaction_safe` attribute.
9

10 A function template that does not specify any of the `transaction_safe`,
11 `transaction_unsafe`, or `transaction_may_cancel_outer` attributes may define a
12 template function that may or may not be implicitly declared safe, depending on whether the body
13 of the template function contains unsafe statements after instantiation. (This feature is especially
14 useful for template libraries, because it allows the use of template library functions within atomic
15 transactions when they are instantiated to contain only safe statements, without requiring these
16 template library functions to be always instantiated to contain only safe statements.) See Section
17 4.3 for an example of such a function template.
18

19 See Section 10 for rules and restrictions on overriding virtual functions declared with the
20 `transaction_safe` attribute.
21

22 When a function pointer of transaction-safe type is assigned or initialized with a value, the
23 initializing or right-hand-side expression must also have transaction-safe type. Furthermore, the
24 transaction safety properties of function pointer parameter types must match exactly. In particular,
25 the type of a function pointer parameter appearing in the type of the target pointer should be
26 transaction-safe if and only if the corresponding parameter type in the initializing or right-hand-
27 side expression is.
28

29 *[Note: An implementation may provide additional mechanisms that make statements safe. Such*
30 *mechanisms might be necessary to implement system libraries that execute efficiently inside*
31 *atomic transactions. Such mechanisms are intended for system library developers and are not*
32 *part of this specification.]*
33

34 The creation (destruction) of an object implicitly invokes a constructor (destructor) function if the
35 object is of a class type that defines a constructor (destructor). The constructor and destructor
36 functions of a class must therefore have transaction-safe or may-cancel-outer type if the
37 programmer intends to allow creation or destruction of objects of that class type inside atomic
38 transactions. In the absence of appropriate programmer-defined constructors (destructors), the
39 creation (destruction) of an object may implicitly invoke a compiler-generated constructor
40 (destructor). A compiler-generated constructor (destructor) for a class has a transaction-safe type
41 if the corresponding constructors (destructors) of all the direct base classes and the
42 corresponding constructors (destructors) of all the non-static data members of the class have
43 transaction-safe type. A compiler-generated constructor (destructor) for a class that is not derived
44 from any other class and has no non-static members of class type always has transaction-safe
45 type.
46

47 The assignment to an object invokes a compiler-generated assignment operator if the object
48 belongs to a class that does not define an assignment operator. A compiler-generated
49 assignment operator for a class has transaction-safe type if the corresponding assignment
50 operators for all the direct base classes and the corresponding assignment operators for all the
51 non-static data members of the class have transaction-safe type.
52

53 *[Note: The `transaction_safe` attribute on function and function pointer declarations allows the*
54 *compiler to ensure that functions whose bodies contain unsafe statements are not called inside*
55 *atomic transactions. Any function with external linkage that the programmer intends to be called*

1 *inside atomic transactions in other translation units must be declared with the*
2 *transaction_safe attribute. To allow client code to use libraries inside atomic transactions,*
3 *library developers should identify functions with external linkage that are known and intended to*
4 *contain only safe statements and annotate their declarations in header files with the*
5 *transaction_safe attribute. Similarly, library developers should use the*
6 *transaction_unsafe attribute on functions known or intended to contain unsafe statements.*
7 *The transaction_unsafe attribute specifies explicitly in a function's interface that the function*
8 *may contain unsafe actions and prevents a function from being implicitly declared safe so that*
9 *future implementations of that function can contain unsafe statements. When annotating a*
10 *function with the transaction_unsafe attribute, library developers should specify this attribute*
11 *on both a function declaration and its definition when the declaration and the definition are*
12 *located in separate header files. This enables client code to include such header files in an*
13 *arbitrary order.]*

14
15 [Note: Library users should not circumvent the restrictions imposed by the library interface by
16 merely modifying transaction-related attributes in the library header files. Similar to other changes
17 to a function declaration (such as changing a function return type or type of a function argument),
18 adding, removing or modifying a transaction-related attribute requires re-compilation. Modifying
19 transaction-related attributes in library header files without re-compiling the library may result in
20 undefined behavior.]

21
22 The header files for the C++ standard library should be modified to specify the annotations for the
23 library functions consistent with the safety properties of those functions. Synchronization (that is,
24 operations on locks and C++11 atomic operations) and certain I/O functions in the C++ standard
25 library should not be declared to have transaction-safe type, as such actions could break
26 atomicity of a transaction, that is, appear to interleave with actions of other threads, under the
27 memory model rules specified in this document (Section 4.5).²

28 **4.3 Examples**

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

```
31  
32 [[transaction_safe]] void f();
```

33
34 The following example shows a function implicitly declared safe by its definition:

```
35  
36 int x;  
37 void g()  
38 {  
39     ++x; // body containing only safe statements  
40 }  
41 // g() is implicitly declared safe after this point  
42
```

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

```
45  
46 void test()  
47 {  
48     __transaction_atomic {  
49         f(); // OK because f() is declared transaction-safe using  
50             // the transaction_safe attribute
```

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

```

1         g(); // OK because g() is implicitly declared safe
2     }
3 }
4

```

The following example illustrates combinations of declarations:

```

6
7 void f(); // first declaration of f
8 void f() { ++w; } // OK, definition of f implicitly declares it transaction-safe
9 void f(); // OK, f is still declared transaction-safe
10
11 [[transaction_safe]] void g(); // first declaration of g
12 void g() { ++x; } // OK: transaction_safe attribute optional on definition
13 void g(); // Error: prior declaration has transaction_safe attribute
14
15 void h(); // first declaration of h
16 [[transaction_safe]] void h() {...} // Error: prior declaration has no
17 // transaction_safe attribute
18
19 void k() { ++y; } // OK, first declaration of k is a definition that implicitly declares it safe
20 [[transaction_unsafe]] void k(); // Error: previous declaration of k
21 // implicitly declared it safe
22
23 [[transaction_unsafe]] void l(); // first declaration of l
24 void l() { ++z; }; // OK, this definition does not implicitly declare k safe because of
25 // a prior declaration with the transaction_unsafe attribute
26
27 void m(); // first declaration of m
28 [[transaction_unsafe]] void m(); // OK, first declaration of m
29 // did not declare it transaction-safe
30

```

The following example illustrates transaction-safe function pointers:

```

32
33 [[transaction_safe]] void (*p1)();
34 void (*p2)();
35 void foo();
36
37 p2 = p1; // OK
38 p2 = f; // OK
39 p1 = p2; // Error: p2 is not transaction-safe
40 p1 = foo; // Error: foo is not transaction-safe
41

```

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:

```

45
46 template<class Op>
47 void t(int& x, Op f) { // Transaction-safety properties of t are not known at this point
48     x++; f(x);
49 }
50
51 class A1 {
52 public:
53     // A1::() is declared transaction-safe
54     [[transaction_safe]] void operator()(int& x);

```

```

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

```

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

```

18
19 [[transaction_safe]] void (*p1) (int&);
20 void (*p2) (int&);
21 [[transaction_unsafe]] void u();
22
23 void n(int v) {
24     int total = 0;
25     __transaction_atomic {
26         t(v, p1); // OK, the call is safe
27         t(v, [&](int x) {total += x;}); // OK, the call is safe
28         t(v, p2); // Error, the call is unsafe
29         t(v, [&](int x) {u();}); // Error, the call is unsafe
30     }
31 }

```

32 **4.4 Nesting**

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

36
37 The following example shows an atomic transaction lexically nested within another atomic
38 transaction:

```

39
40 __transaction_atomic {
41     x++;
42     __transaction_atomic {
43         y++;
44     }
45     z++;
46 }
47

```

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

```

50
51 [[ transaction_safe ]] void bar()
52 {
53     __transaction_atomic { x++; }
54 }

```

```
1
2  __transaction_atomic {
3     bar();
4 }
5
```

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

8 **4.5 Memory model**

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

17 **5. Transaction expressions**

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

```
22
23     __transaction_relaxed ( expression )
24     __transaction_atomic ( expression )
25
```

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

```
28
29     __transaction_atomic { T temp = expression ; }
30
```

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

33
34 A transaction expression can be used to evaluate an expression in a transaction. This is
35 especially useful for initializers, as illustrated by the following example:

```
36
37     SomeObj myObj = __transaction_atomic ( expr ); // calls copy constructor
38
```

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

43
44 [Note: A *transaction expression on an initializer applies only to evaluating the initializer. The*
45 *initialization (for example, executing a copy constructor) is performed outside of a transaction.*
46 *Transaction expressions and statements thus do not allow a programmer to specify that the*
47 *initialization statement should be executed inside a transaction without restricting the scope of the*
48 *initialized object.*]

49
50 A transaction expression cannot contain a transaction statement, a cancel statement (Section 8)
51 or a cancel-and-throw statement (Section 9) since the C++ standard does not allow expressions
52 to contain statements.

1
2 Implementations that support statement-expressions could syntactically allow a cancel statement
3 or a cancel-and-throw statement to appear within a transaction expression. However, a cancel or
4 cancel-and-throw statement must not appear inside a transaction expression unless the cancel or
5 cancel-and-throw statement is either annotated with the `outer` attribute or is lexically enclosed
6 within an atomic transaction statement that is lexically enclosed within that transaction expression.

7 **6. Function transaction blocks**

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

```
10  
11 void f() __transaction_relaxed {  
12     // body of f() executes in an relaxed transaction  
13 }  
14  
15 void g() __transaction_atomic {  
16     // body of g() executes in a atomic transaction  
17 }  
18
```

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

21
22 A function transaction block on a constructor causes the constructor body and all member and
23 base class initializers of that constructor to execute inside a transaction. The function transaction
24 block syntax thus allows programmers to include member and base class initializers in
25 constructors in a transaction. In the following example, the constructor `Derived()` and its
26 initializers all execute atomically:

```
27  
28 class Base {  
29     public:  
30         Base(int id) : id_(id) {}  
31     private:  
32         const int id_;  
33 };  
34  
35 class Derived : public Base {  
36     public:  
37         Derived() __transaction_atomic : Base(count++) { ... }  
38     private:  
39         static int count = 0;  
40 };  
41
```

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

48
49 A function transaction block can be combined with the function try block syntax. If the
50 `__transaction_atomic` or the `__transaction_relaxed` keyword appears before the `try`
51 keyword, the catch block is part of the function transaction block. If the transaction keyword
52 appears after the `try` keyword, the catch block is not part of the function transaction block:
53

```

1 Derived::Derived()
2 try __transaction_atomic : Base(count++) {}
3 catch (...) {} // catch is not part of transaction
4
5 Derived::Derived()
6 __transaction_atomic try : Base(count++) { ... }
7 catch (...) {} // catch is part of transaction
8

```

[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:

```

15 // error: a relaxed transaction is never transaction-safe
16 [[transaction_safe]] void f() __transaction_relaxed { ... }
17 ]
18 ]
19

```

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 specification³

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:

```

31 __transaction_atomic noexcept [ (constant-expression) ] compound-statement
32 __transaction_atomic noexcept [ (constant-expression) ] ( expression )
33
34 __transaction_relaxed noexcept [ (constant-expression) ] compound-statement
35 __transaction_relaxed noexcept [ (constant-expression) ] ( expression )
36

```

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:

```

42 __transaction_atomic [[ outer ]] noexcept
43 [ ( constant-expression ) ] compound-statement
44

```

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.

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

3
4 The following example declares a transaction statement that does not allow an exception to
5 propagate outside of its scope:

```
6  
7 __transaction_atomic noexcept (true) compound-statement  
8 __transaction_atomic noexcept compound-statement
```

9
10 A transaction statement (expression) that does not include a `noexcept` specification or includes a
11 `noexcept` specification that has a constant-expression that evaluates to `false` may throw an
12 exception that escapes the scope of the transaction statement (expression).

13
14 The following example declares a transaction statement that allows an exception to propagate
15 outside of its scope:

```
16  
17 __transaction_atomic noexcept(false) compound-statement  
18 __transaction_atomic compound-statement
```

19
20 [Note: Omitting a `noexcept` specification on a transaction statement (expression) that may throw
21 an exception makes it easy to overlook the possibility that an exception thrown from within the
22 dynamic extent of that statement (expression) can result in the statement (expression) being only
23 partially executed. Therefore, programmers are strongly encouraged to explicitly state whether
24 exceptions can be thrown from transaction statements (expressions) by using `noexcept`
25 specifications. We considered an alternative approach in which the absence of a `noexcept`
26 specification is interpreted as if a `noexcept(true)` clause were present, which makes
27 mandatory an explicit `noexcept(false)` specification on a transaction statement (expression) that
28 may throw an exception. However, such an interpretation would be inconsistent with the existing
29 rules for `noexcept` specifications on function declarations.]

30
31 A `noexcept` specification is not allowed on a function transaction block as such a specification is
32 redundant with a `noexcept` specification on a function declaration (that is, a `noexcept` specification
33 that may appear before the `__transaction_atomic` or `__transaction_relaxed` keyword
34 denoting a function transaction block).

35 8. Cancel statement

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

```
40  
41 __transaction_atomic {  
42     stmt1  
43     __transaction_cancel;  
44 }  
45 stmt2
```

46
47 In its basic form (that is, without the `outer` attribute), a cancel statement rolls back all side
48 effects of the immediately enclosing atomic transaction statement (that is, the smallest atomic
49 transaction statement that encloses the cancel statement) and transfers control to the statement
50 following the transaction statement. Thus, in the example above the cancel statement undoes the
51 side effects of `stmt1` and transfers control to `stmt2`.

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

6
7 *[Note: A cancel statement applies only to atomic transaction statements (including outer atomic
8 transaction statements). A cancel statement cannot be used to roll back a function transaction
9 block or a transaction expression, unless that block or expression is rolled back as part of rolling
10 back an atomic transaction statement.]*

11 **8.1 The outer attribute on cancel statements**

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

```
13  
14 __transaction_cancel [[ outer ]];  
15
```

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

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

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

32 The cancel-outer statement provides a convenient way to cancel an outermost atomic transaction
33 from anywhere within its dynamic extent. For example, when an error is encountered, the
34 programmer can cancel all transactions from the most nested transaction to the outer transaction.
35 The 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**

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

1
2 If a class type has constructors or a destructor with may-cancel-outer type, then objects of that
3 type must be declared so as to ensure that the affected constructor or destructor is invoked within
4 the dynamic scope of an outer atomic transaction. Declarations of such an object leading to the
5 invocation of the affected constructor or destructor should appear within the lexical scope of an
6 outer atomic transaction or in a may-cancel-outer function. Moreover, an object should be
7 declared in such a way that the affected constructor or destructor is invoked in the same scope as
8 the declaration. For example, if a class has a constructor with may-cancel-outer type then a
9 program may not contain global or static declarations of that type resulting in the invocation of the
10 affected constructor. If a class has a destructor with may-cancel-outer type then a program may
11 not contain global, static, function-local static or thread_local declarations of that type.
12

13 Like the `transaction_safe` attribute, the `transaction_may_cancel_outer` attribute
14 specifies a property of the type of the declared function or function pointer, which is propagated
15 along with the type. Just like `transaction_safe`, it may be meaningfully used in `typedef`
16 declarations.
17

18 A pointer to a function with transaction-safe type may be implicitly converted to a pointer to a
19 function with may-cancel-outer type (or to a pointer to a function with transaction-unsafe type).
20 Such conversions have no effect on the ranking of conversions sequences. A pointer to a may-
21 cancel-outer function is not implicitly convertible to a pointer to a non-may-cancel-outer function.
22 Allowable function pointer conversions:
23



26
27 A function or function pointer must not be declared with both the
28 `transaction_may_cancel_outer` and `transaction_safe` attributes. A function must not
29 be declared with both the `transaction_may_cancel_outer` and `transaction_unsafe`
30 attributes. That is, a function declaration, a function pointer declaration, or multiple declarations of
31 one function must not specify both attributes. If any declaration of a function specifies the
32 `transaction_may_cancel_outer` attribute then every declaration of that function (except its
33 definition, if it is not a virtual function) must specify the `transaction_may_cancel_outer`
34 attribute, and the first declaration must do so even if it is the definition of a non-virtual function.
35 The `main` function must not be declared with the `transaction_may_cancel_outer` attribute.
36 A function may be declared with both `transaction_may_cancel_outer` and
37 `transaction_callable` attributes.
38

39 A function call to a function declared with the `transaction_may_cancel_outer` attribute
40 (before the function call) is a safe statement (Section 4.2). A function call through a function
41 pointer that was declared with the `transaction_may_cancel_outer` attribute is also a safe
42 statement. The body of a function declared with the `transaction_may_cancel_outer`
43 attribute must not contain unsafe statements.
44

45 See Section 10 for rules and restrictions on overriding virtual functions declared with the
46 `transaction_may_cancel_outer` attribute.
47

48 When a function pointer declared with the `transaction_may_cancel_outer` attribute is
49 assigned or initialized with a value, that value must be a pointer to a function of transaction-safe
50 or may-cancel-outer type. When a function pointer declared without the
51 `transaction_may_cancel_outer` attribute is assigned or initialized with a value, that value
52 must not be a pointer to a function of may-cancel-outer type. As in the case of the
53 `transaction_safe` attribute, parameter types for function pointers assigned in this way must
54 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
<pre> __transaction_atomic { x++; __transaction_cancel; } </pre>	<pre> x = 1; </pre>

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.

1
2 In an exception handler of integral or enumerated type, the cancel-and-throw statement may
3 optionally leave out the exception expression, in which case the specified exception is the current
4 exception.
5

6 A cancel-and-throw statement has the same properties with respect to the memory model as a
7 cancel statement (Section 8.4): In a data-race-free program, a transaction cancelled by a cancel-
8 and-throw statement has no visible side effects. However, the individual operations of a
9 transaction that executed before the transaction was cancelled are part of the program execution
10 and may contribute to data races.
11

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

17 **9.1 The outer attribute on cancel-and-throw statements**

18 The cancel-and-throw statement may be annotated with the `outer` attribute, in which case it is a
19 *cancel-outer-and-throw* statement:
20

```
21 __transaction_cancel [[ outer ]] throw expropt;
```

22
23 A cancel-outer-and-throw statement operates in the same way as a cancel-and-throw statement
24 except that it rolls back the side effects of the outer atomic transaction that dynamically contains it
25 and throws the exception from the outer atomic transaction. Like the cancel-outer statement, a
26 cancel-outer-and-throw statement need not be enclosed within the lexical scope of an atomic
27 transaction, but it must appear either within the lexical scope of an outer atomic transaction or in
28 a may-cancel-outer function.

29 **9.2 Examples**

30 An unannotated cancel-and-throw statement rolls back the side effects of only its immediately
31 enclosing atomic transaction. In the following example, the cancel-and-throw statement rolls back
32 *stmt2* but not *stmt1*, and the thrown exception `1` propagates out of the outermost atomic
33 transaction:
34

```
35 bool flag1 = false, flag2 = false;  
36 try {  
37     __transaction_atomic {  
38         flag1 = true; // stmt1  
39         __transaction_atomic {  
40             flag2 = true; // stmt2  
41             __transaction_cancel throw 1;  
42         }  
43     }  
44 } catch(int& e) {  
45     assert(flag1 == true && flag2 == false);  
46 }  
47
```

48 A cancel-outer-and-throw statement rolls back the side effects of the outer atomic transaction that
49 dynamically contains it. In the following example, the cancel-outer-and-throw statement rolls
50 back both *stmt1* and *stmt2*, after which the thrown exception `1` propagates out of the outer atomic
51 transaction (which is the outermost atomic transaction):
52

```

1  bool flag1 = false, flag2 = false;
2  try {
3      __transaction_atomic [[outer]] {
4          flag1 = true; // stmt1
5          __transaction_atomic {
6              flag2 = true; // stmt2
7              __transaction_cancel [[outer]] throw 1;
8          }
9      }
10 } catch(int& e) {
11     assert(flag1 == false && flag2 == false);
12 }

```

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:

```

23 try {
24     __transaction_atomic {
25         try {
26             throw 1;
27         } catch(int& e) {
28             cout << "Caught e!" << endl;
29         }
30     }
31 } catch (int& e) {
32     assert(0); // never reached!
33 }

```

Example 2:

```

37 try {
38     __transaction_atomic {
39         try {
40             __transaction_cancel throw 1;
41         } catch(int& e) {
42             assert(0); // never reached!
43         }
44     }
45 } catch (int& e) {
46     cout << "Caught e!" << endl;
47 }

```

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:

```

53 class X { int x;};
54
55 __transaction_atomic noexcept(false) {

```

```
1     __transaction_cancel throw X(); // Error: X() is of class type
2 }
3
```

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

```
7
8     __transaction_atomic noexcept(false) {
9         try {
10             stmt
11         } catch (int&) {
12             __transaction_cancel throw;
13         }
14     }
15
```

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

```
19
20     __transaction_atomic noexcept(false) {
21         try {
22             stmt
23         } catch (...) {
24             __transaction_cancel throw; // Error: current exception may be of any type
25         }
26     }
```

27 10. Inheritance and compatibility rules for attributes

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

- 35 • A virtual function of transaction-safe type may be overridden only by a virtual function of
36 transaction-safe type.
- 37 • A virtual function of may-cancel-outer type can be overridden only by a virtual function of
38 either may-cancel-outer or transaction-safe type.
- 39 • A virtual function of may-cancel-outer type may override only a virtual function of may-
40 cancel-outer type.
- 41 • Any function pointer type appearing in a signature of an overriding function must have the
42 same transactional attributes as the corresponding function pointer type in the signature
43 of the overridden function.

44
45 The following example illustrates the class inheritance rules for transaction-related function
46 attributes:

```
47
48     class C {
49     public:
50         [[transaction_safe]] void f();
51         [[transaction_safe]] virtual void v();
52         [[transaction_unsafe]] virtual void w();
53     };
54
```

```

1  class D : public C {
2  public:
3      void f();           // OK: D::f redefines C::f
4      virtual void v();  // Error: D::v overrides C::v; needs transaction_safe
5      virtual void w();  // OK: transaction_unsafe on D::w is optional
6      using C::v;        // OK: C::v preserves the transaction_safe attribute
7  };
8
9

```

10 **11. Class attributes**

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

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

```

22
23 class C [[transaction_safe]] {
24     void f();           // declared as transaction_safe
25     virtual void v();   // declared as transaction_safe
26     [[transaction_unsafe]] virtual void w(); // declared as
27                                     // transaction_unsafe
28 };
29

```

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

33

1 **Appendix A. Grammar**

2 *atomic transaction-statement:*

3 `__transaction_atomic txn-outer-attributeopt txn-noexcept-specopt compound-`
4 `statement`

6 *relaxed transaction-statement:*

7 `__transaction_relaxed txn-noexcept-specopt compound-statement`

9 *atomic transaction-expression:*

10 `__transaction_atomic txn-noexcept-specopt (expression)`

12 *relaxed transaction-expression:*

13 `__transaction_relaxed txn-noexcept-specopt (expression)`

15 *atomic function-transaction-block:*

16 *atomic-basic-function-transaction-block*

17 `__transaction_atomic basic-function-try-block`

19 *relaxed function-transaction-block:*

20 *relaxed basic-function-transaction-block*

21 `__transaction_relaxed basic-function-try-block`

23 *atomic basic-function-transaction-block*

24 `__transaction_atomic ctor-initializeropt compound-statement`

26 *relaxed basic-function-transaction-block*

27 `__transaction_relaxed ctor-initializeropt compound-statement`

29 *cancel-statement:*

30 `__transaction_cancel txn-outer-attributeopt ;`

32 *cancel-and-throw-statement:*

33 `__transaction_cancel txn-outer-attributeopt throw-expression ;`

35 *txn-noexcept-spec:*

36 *noexcept-specification*

38 *txn-outer-attribute:*

39 `[[outer]]`

41 *postfix-expression:*

42 */* ... existing C++11 rules ... */*

43 *transaction-expression*

44

```

1  statement:
2      /* ... existing C++11 rules ... */
3      attribute-specifieropt transaction-statement
4
5  jump-statement:
6      /* ... existing C++11 rules ... */
7      cancel-statement
8      cancel-and-throw-statement
9
10 function-body:
11     /* ... existing C++11 rules ... */
12     function-transaction-block
13
14 function-try-block:
15     basic-function-try-block
16     try basic-function-transaction-block handler-seq
17
18 basic-function-try-block:
19     /* ... existing C++11 rules for function-try-block ... */

```

20 **Appendix B. Feature dependences**

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

24
25 As general guidance, we recommend that an implementation that does not support a certain
26 feature accepts the syntax of that feature and issues an informative error message, preferably
27 indicating that the feature is not supported by the implementation but is a part of the specification.

28
29 The language features described in this specification are interdependent. Eliminating a certain
30 feature may make some other features unusable. For example, without the outer atomic
31 transactions, the cancel-outer statement is unusable; that is, it is not possible to write a legal
32 program that executes a cancel-outer statement and does not contain an outer atomic transaction
33 statement (because the cancel-outer statement must execute within the dynamic extent of an
34 outer atomic transaction). Some other features may remain usable but become irrelevant. For
35 example, without atomic transactions, the `transaction_safe` attribute can occur in legal
36 programs but serves no purpose. We recommend that an implementation that chooses to support
37 a certain irrelevant feature issues an informative warning specifying that the feature is supported
38 for compatibility purposes but has no effect. In the rest of this section, we describe dependences
39 between the features and identify the consequences of omitting a particular feature or
40 combination of features.

41
42 **Transaction statements, transaction expressions and function transaction blocks.** This
43 specification provides three language constructs for specifying transactions: transaction
44 statements, transaction expressions and function transaction blocks. All other features described
45 in this specification are dependent on the presence of at least one of these constructs. Therefore
46 any implementation should include at least one of these constructs. The constructs themselves
47 are independent of each other. An implementation may include one, two or all three of them.

48
49 All three constructs allow for specifying two forms of transactions – relaxed transactions and
50 atomic transactions. Furthermore, atomic statements may be annotated with the `outer` attribute

1 to indicate that they execute as outer atomic transactions. These forms of transactions are
2 independent of each other. An implementation may include either relaxed transactions, or atomic
3 transactions, or both. It may also choose not to support outer atomic transactions, or to require all
4 atomic transactions to be outer atomic transactions.
5

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

- 9 • The concept of safe and unsafe statements and the `transaction_safe` and
10 `transaction_unsafe` function attributes are irrelevant without atomic transactions
11 (because the safety concept and attributes are used to impose restrictions on statements that
12 can be executed within an atomic transaction).
- 13 • The cancel statement is unusable without atomic transaction statements (because it applies
14 only to atomic transaction statements).
- 15 • The cancel-and-throw statement is unusable unless an implementation supports either
16 atomic transaction statements or atomic function transaction blocks (because it applies only
17 to atomic transaction statements or atomic function transaction blocks).
- 18 • The cancel-outer statement, the cancel-outer-and-throw statement, and the
19 `transaction_may_cancel_outer` attribute are unusable without outer atomic
20 transactions (because the cancel-outer statements, cancel-outer-and-throw statements and
21 calls to functions declared with the `transaction_may_cancel_outer` attribute can
22 execute only within the dynamic extent of an outer atomic transaction).
23

24 The only feature used solely in conjunction with relaxed transactions is the
25 `transaction_callable` attribute. This attribute is irrelevant without relaxed transactions
26 (because it indicates that a function might be called within a relaxed transaction).
27

28 An implementation may impose additional restrictions on nesting of various forms of transactions
29 without affecting the rest of the specified features.
30

31 **Function call safety.** This specification includes three features related to the safety of function
32 calls – the `transaction_safe` and `transaction_unsafe` attributes and the concept of
33 functions being implicitly declared safe. Eliminating one or more of the function call safety
34 features does not affect the rest of the specification. However, different combinations of these
35 features offer different degrees of ability to call functions from within atomic transactions:
36

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

1
2 An implementation may include the `transaction_safe` attribute for function declarations, or
3 function pointer declarations, or both. An implementation that does not support the
4 `transaction_safe` attribute for function pointer declarations must disallow calls via function
5 pointers inside atomic transactions.
6

7 **Cancel and cancel-and-throw statements.** This specification provides two forms of a cancel
8 statement – a basic cancel statement that cancels the immediately enclosing atomic transaction
9 and the cancel-outer statement that cancels the enclosing outer atomic transaction. This
10 specification also provides two similar forms of a cancel-and-throw statement – a basic cancel-
11 and-throw statement and the cancel-and-throw-outer statement. The cancel and cancel-and-
12 throw statements and the two forms of each statement are independent of each other. An
13 implementation may include any combination of these statements and their forms. Eliminating
14 either the basic cancel statement or the basic cancel-and-throw statement does not affect the rest
15 of the specification. Eliminating either the cancel-outer statement or the cancel-outer-and-throw
16 statement, but not both of these statements, also does not affect the rest of the features.
17 Eliminating both the cancel-outer statement and the cancel-outer-and-throw statement makes the
18 `transaction_may_cancel_outer` attribute irrelevant (because this attribute is used to specify
19 that a function may contain either the cancel-outer or cancel-outer-and-throw statement in its
20 dynamic scope) and limits the usability of the `outer` attribute on transaction statements (because
21 the main purpose of this attribute is to specify atomic transactions that can be cancelled by the
22 cancel-outer or cancel-outer-and-throw statement). The `outer` attribute, however, still can be
23 used to specify that an atomic transaction statement cannot be nested within another atomic
24 transaction.
25

26 **The `transaction_may_cancel_outer` attribute.** Eliminating the
27 `transaction_may_cancel_outer` attribute reduces the usability of the cancel-outer and
28 cancel-outer-and-throw statements. An implementation that does not support this attribute must
29 not allow the cancel-outer and the cancel-outer-and-throw statements outside of the lexical scope
30 of an outer atomic transaction statement (because the implementation has no ability to verify that
31 a function containing a cancel-outer statement in its dynamic scope is not called outside of an
32 outer atomic transaction).
33

34 **The `transaction_callable` attribute.** This attribute has no semantic meaning: it is only a
35 hint to the compiler that certain optimizations might be worthwhile. Eliminating this attribute has
36 no effect on other features.
37

38 **noexcept specification.** A noexcept specification facilitates development of more reliable
39 programs. Not supporting noexcept specifications on transaction statements and/or expressions
40 has no effect on other features.
41

42 **Exceptions.** An implementation that implements a subset of this specification may choose to
43 provide limited support for exceptions inside transactions (including the exceptions thrown by the
44 throw statement and/or exceptions thrown by the cancel-and-throw statement). For example, an
45 implementation might disallow throwing an exception from within code that could be executed
46 within a transaction, or disallow exceptions from escaping the scope of a transaction. Such
47 restrictions might make noexcept specifications irrelevant.
48

49 **Unsafe statements.** This specification defines certain statements as unsafe. An implementation
50 that implements a subset of this specification might choose to treat additional statements as
51 unsafe. For example, an implementation might choose to treat built-in `new` and `delete` operators
52 as unsafe and disallow them inside atomic transactions. We suggest that such an implementation
53 provides a workaround to allow programmers to allocate and deallocate objects within atomic
54 transactions, and indicate this in an error message produced when encountering a `new` or

1 delete built-in operator in an atomic transaction. In most cases, treating additional statements
2 as unsafe would not affect the rest of the specification.
3

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

7 Appendix C. Extensions

8 **Allowing unsafe statements inside atomic transactions.** To relax the restriction of statically
9 disallowing unsafe statements inside atomic transactions and functions declared with the
10 `transaction_safe` or `transaction_may_cancel_outer` attribute, we could make
11 executing such statements a dynamic error that rolls back the atomic transaction and then either
12 throws an exception or sets an error code. However, this approach would forgo the benefits of
13 compile-time checking and instead shift the burden of detecting and handling atomic transactions
14 that executed unsafe operations to a programmer.
15

16 **Transaction declaration statements.** The features described in this specification do not allow
17 executing an initialization statement inside a transaction without changing the scope of the
18 initialized object (Section 5). We could introduce a transaction declaration statement that causes
19 all the actions initiated by the initialization statement to be performed inside a transaction. A
20 transaction declaration statement would be specified by placing the `__transaction_relaxed`
21 or the `__transaction_atomic` keyword before the declaration as illustrated by the following
22 example, where both the copy constructor and evaluation of its argument are executed within a
23 transaction:
24

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

27
28 **Relaxing the lexical scope restriction.** We could remove the lexical scoping restriction on
29 cancel statements without `outer` attribute so that such statements could appear anywhere inside
30 the dynamic scope of an atomic transaction. Rollbacks don't make sense outside of the dynamic
31 scope of an atomic transaction, however, so we could define such cancel statements such that
32 they are either a runtime or compile-time error. In the former case, we could define cancel
33 statements executed outside the dynamic scope of an atomic transaction as leading to a runtime
34 failure that terminates the program (similar to a re-throw outside of the dynamic scope of a catch
35 block); for example, by providing a `cancel()` API call that fails if called outside the dynamic
36 scope of an atomic transaction. To support the latter case, we could introduce a new function
37 attribute (e.g., the `transaction_atomic_only` attribute) specifying that a function can only be
38 called within the dynamic extent of an atomic transaction because it may execute a cancel
39 statement outside the lexical scope of an atomic transaction; thus an unannotated
40 `__transaction_cancel` statement must appear within the lexical scope of either an atomic
41 transaction or a properly-declared function (that is, a function declared with the
42 `transaction_atomic_only` or `transaction_may_cancel_outer` attribute). Similar to
43 lexical scoping, this has the advantage that the implementation can distinguish atomic
44 transactions that require rollback. Note, that although an unannotated cancel statement may
45 appear in a function declared with the `transaction_may_cancel_outer` attribute, using a
46 single attribute for functions that may contain an unannotated cancel statement and functions that
47 may contain a cancel-outer statement is not a good idea; such a design decision would artificially
48 restrict the usage of unannotated cancel statements to the dynamic scope of an outer atomic
49 transaction.
50

51 **Supporting cancelling of relaxed transactions.** Allowing cancel statements only inside atomic
52 transactions limits combinations of irrevocable actions and cancel statements to well-structured
53 programming patterns (such as an atomic-within-relaxed idiom in Section 8.3). Alternatively, we

1 could allow arbitrary syntactic combinations of cancel statements and irrevocable actions and
2 place the burden of preventing dynamically unsafe combinations on a programmer. That is, we
3 could allow a cancel statement to appear anywhere within the scope of a relaxed transaction and
4 require that programmers not to use `__transaction_cancel` after a call to an irrevocable
5 action (i.e., any call to an unsafe statement). In this case, cancelling a relaxed transaction that
6 executed an irrevocable action would be a run-time failure that exits the program with an error.
7 We could also devise static rules that avoid rollback after an irrevocable action at the expense of
8 prohibiting some dynamically safe combinations of cancel statements and irrevocable actions.
9

10 With this change, we could also forgo differentiating between relaxed and atomic transactions
11 and simply treat relaxed transactions that contain only safe statements as atomic transactions.
12 However, we believe that supporting statically enforced atomic transactions encourages the
13 development of more robust and reliable software by allowing the programmer to declare the
14 intention that a block of code should appear atomic (with the corresponding restriction that it
15 should contain only safe operations). Effectively, atomic transactions act as a compile-time
16 assertion that allows atomicity violations to be identified at compile time rather than run time.
17

18 **Adding an else clause to atomic transaction statements.** We could add an else-clause to
19 “catch” cancels. For example:

```
20  
21     __transaction_atomic {  
22         stmt  
23     } else {  
24         // control ends up here if stmt cancels the transaction  
25     }  
26
```

27 The else-clause allows the programmer to determine whether an atomic transaction cancelled
28 without resorting to explicit flags. We could also use the else-clause to provide alternate actions
29 in case the atomic transaction attempts to execute an unsafe statement, relaxing the rule that
30 prohibits unsafe function calls inside the dynamic scope of an atomic transaction. Thus, an
31 attempt to execute an unsafe statement inside an atomic transaction would rollback the statement
32 and transfer control to the else-clause.
33

34 **Introducing a retry statement.** We could define a retry statement (e.g.,

```
35 __transaction_retry)
```

36 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
37 the outer atomic transaction is within the dynamic extent of a relaxed transaction, however, will
38 result in an infinite loop (relaxed transactions are serializable with respect to atomic transactions
39 thus re-execution will follow the same path) and may prevent other transactions from making
40 progress (depending on implementation). It might be possible to statically disallow outer atomic
41 transactions from nesting inside a relaxed transaction using additional function attributes, but this
42 might unnecessarily restrict use of code that might execute outer atomic transactions and it
43 introduces a function attribute that might propagate all over the program.
44

45 **Removing restrictions on types of exceptions thrown by the cancel-and-throw statement.**

46 This specification requires exceptions thrown by the cancel-and-throw statement to be of integral
47 or enumerated types. We could remove this restriction and allow the cancel-and-throw statement
48 to throw exceptions of arbitrary types. This, however, could lead to subtle hard-to-detect bugs
49 when an exception object contains or refers to the state that is not meaningful after the
50 transaction is cancelled. For example, if an exception object points to an object allocated inside a
51 transaction, that object would be deallocated when the transaction is cancelled, resulting in a
52 dangling pointer. If an exception object contains a pointer to an object allocated outside of the
53 transaction, throwing this object can still lead to an inconsistent state if the pointer is implemented
54 as a shared pointer with reference count. When transaction is cancelled the increment of the
55 reference count would be undone, possibly causing the thrown object to unexpectedly disappear

1 due the reference count being one too low. Finally, a thrown object may contain inconsistent state
2 even if it contains no pointers. For example, if the thrown object is an instance of a class T,
3 whose constructors and destructors keep track of all instances of T, the tracking of that object is
4 going to be lost after the transaction is cancelled.
5

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

10 **Region attributes.** We could introduce region attributes that act as default attributes for functions
11 declared within a region of code. This would allow the programmer to annotate multiple function
12 declarations by specifying the attribute only once. For example, a programmer could annotate all
13 declarations in a header file as `transaction_safe`, by including them in a code region
14 annotated with the `transaction_safe` attribute.
15
16

17 **Appendix D. Changes compared to version 1.0**

18 This specification contains the following changes compared to its previous version – the Draft
19 Specification of Transactional Language Constructs for C++, version 1.0:
20

21 **Transaction keywords.** The `__transaction` keyword and its associated attributes, `atomic`
22 and `relaxed`, have been replaced by the `__transaction_atomic` and
23 `__transaction_relaxed` keywords, respectively. Previously, an atomic transaction could be
24 declared by using just the `__transaction` keyword, while a relaxed transaction required the
25 `__transaction` keyword annotated with the `[[relaxed]]` attribute. The new syntax puts
26 relaxed and atomic transactions on equal footing, by providing each with its own keyword.
27

28 **Transactional types.** The transactional function properties defined by `transaction_safe`,
29 `transaction_unsafe`, and `transaction_may_cancel_outer` attributes are now part of a
30 function type. As such, these properties might be specified in typedef declarations and
31 propagated as part of the type. They are still ignored, however, for the overload resolution.
32 Previously, the transactional properties of a function had many characteristics of type without
33 being such, which limited their applicability (e.g., they could not participate in typedef
34 declarations) and left the behavior in multiple corner cases unspecified. Elevating transactional
35 function properties to types solves these problems.
36

37 **Exception specifications and `noexcept` specifications.** The specification now supports
38 C++11's `noexcept` specifications and has removed support for C++11's deprecated exception
39 specifications. This was done because exception specifications have been deprecated in C++11
40 and have been replaced by `noexcept` specifications.
41

42 **Cancel-and-throw exception types.** The types of exceptions thrown by cancel-and-throw are
43 now limited to integral and enumeration types. This change was made to prevent subtle bugs due
44 to destroyed transactional state escaping the scope of the transaction via an exception object.
45

46 **Memory model.** The memory model now includes complete rules on how `TransactionStart` and
47 `TransactionEnd` operations contribute to the “sequenced-before” relationship.
48

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