Contracts for C++

TIMUR DOUMLER

Contracts for C++

Version 1.0

(pre-Wrocław Edition)

Timur Doumler

CppCon
18 September 2024

The Swan, The Pike, and The Crab
– Fable by Ivan Krylov, 1814

Proposal: add contract assertions to C++

```
Widget getWidget (index i)
  pre (i > 0)                       // precondition assertion
  post (w: w.index() == i);         // postcondition assertion
```

Proposal: add contract assertions to C++

Function contract assertions

```
Widget getWidget (index i)
  pre (i > 0)                    // precondition assertion
  post (w: w.index() == i);      // postcondition assertion
```

Proposal: add contract assertions to C++

```
Widget getWidget (index i)
  pre (i > 0)                       // precondition assertion
  post (w: w.index() == i)          // postcondition assertion
{
  auto* db = getDatabase();
  contract_assert (db != nullptr);  // assertion statement
  return db->retrieveWidget (i);
}
```

Contracts for C++

Document #:	P2900R8
Date:	2024-07-26
Project:	Programming Language C++
Audience:	EWG, LEWG

Reply-to: Joshua Berne <jberne4@bloomberg.net>
Timur Doumler <papers@timur.audio>
Andrzej Krzemieński <akrzemi1@gmail.com>
— with —
Gašper Ažman <gasper.azman@gmail.com>
Louis Dionne <ldionne@apple.com>
Tom Honermann <tom@honermann.net>
John Lakos <jlakos@bloomberg.net>
Lisa Lippincott <lisa.e.lippincott@gmail.com>
Jens Maurer <jens.maurer@gmx.net>
Ryan McDougall <mcdougall.ryan@gmail.com>
Jason Merrill <jason@redhat.com>
Ville Voutilainen <ville.voutilainen@gmail.com>

Abstract

In this paper, we propose a Contracts facility for C++ that has been carefully considered by SG21 with a high bar set for level of consensus. The proposal includes syntax for specifying three kinds of contract assertions: precondition assertions, postcondition assertions, and assertion statements. In addition, we specify four evaluation semantics for these assertions — one non-checking semantic, *ignore*, and three checking semantics, *observe*, *enforce*, and *quick_enforce* — as well as the ability to specify a user-defined handler for contract violations. The features proposed in this paper allow C++ users to leverage contract assertions in their ecosystems in numerous ways.

Contracts for C++

Document #: P2900R8
Date: 2024-07-26
Project: Programming Language C++
Audience: EWG, LEWG

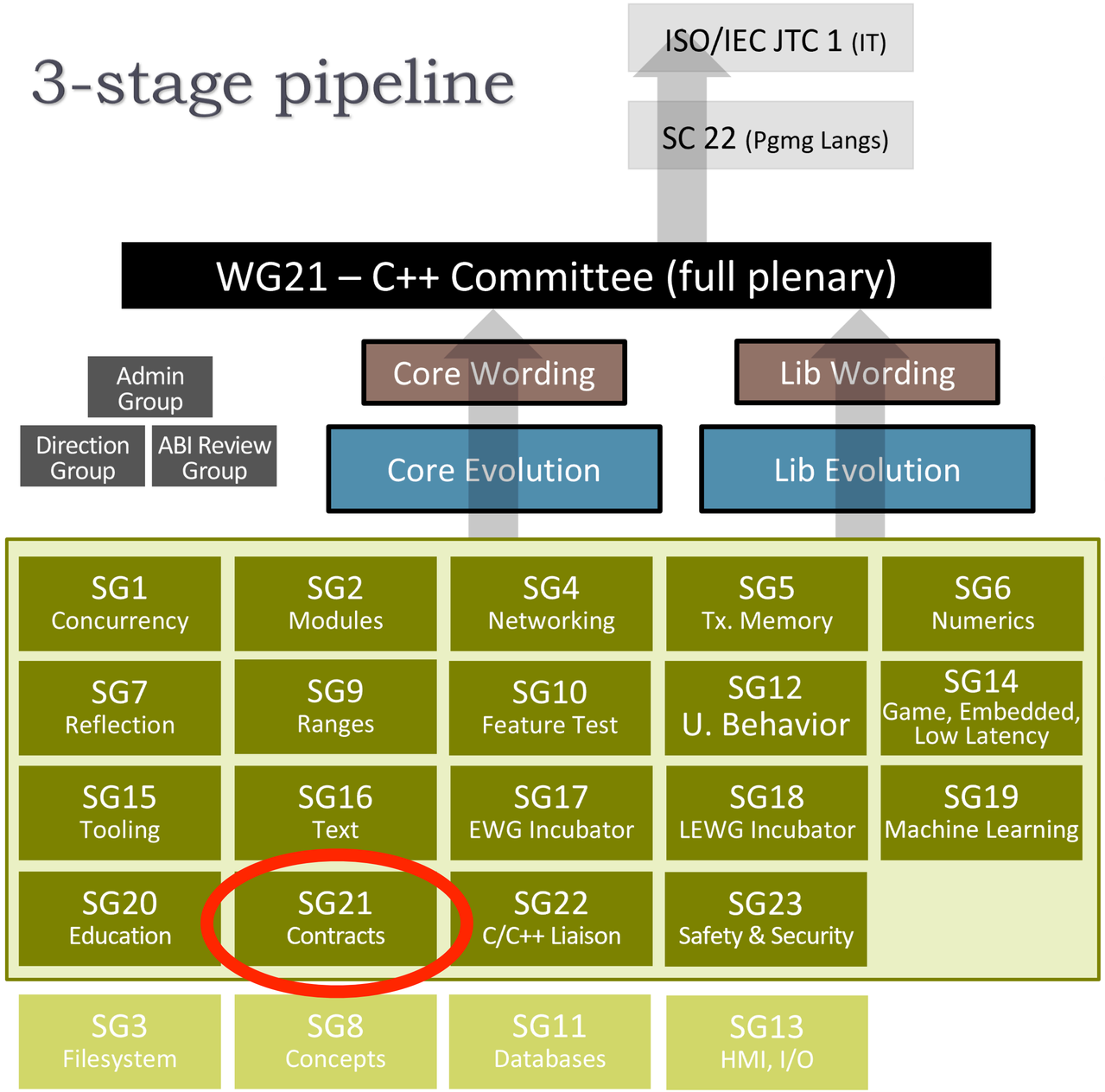
Reply-to: Joshua Berne <jberne4@bloomberg.net>
Timur Doumler <papers@timur.audio>
Andrzej Krzemieński <akrzemi1@gmail.com>
— with —
Gašper Ažman <gasper.azman@gmail.com>
Louis Dionne <ldionne@apple.com>
Tom Honermann <tom@honermann.net>
John Lakos <jlakos@bloomberg.net>
Lisa Lippincott <lisa.e.lippincott@gmail.com>
Jens Maurer <jens.maurer@gmx.net>
Ryan McDougall <mcdougall.ryan@gmail.com>
Jason Merrill <jason@redhat.com>
Ville Voutilainen <ville.voutilainen@gmail.com>

https://wg21.link/p2900

Abstract

In this paper, we propose a Contracts facility for C++ that has been carefully considered by SG21 with a high bar set for level of consensus. The proposal includes syntax for specifying three kinds of contract assertions: precondition assertions, postcondition assertions, and assertion statements. In addition, we specify four evaluation semantics for these assertions — one non-checking semantic, *ignore*, and three checking semantics, *observe*, *enforce*, and *quick_enforce* — as well as the ability to specify a user-defined handler for contract violations. The features proposed in this paper allow C++ users to leverage contract assertions in their ecosystems in numerous ways.

3-stage pipeline



(F)DIS Approval

CD & PDTS Approval

Internal Approval

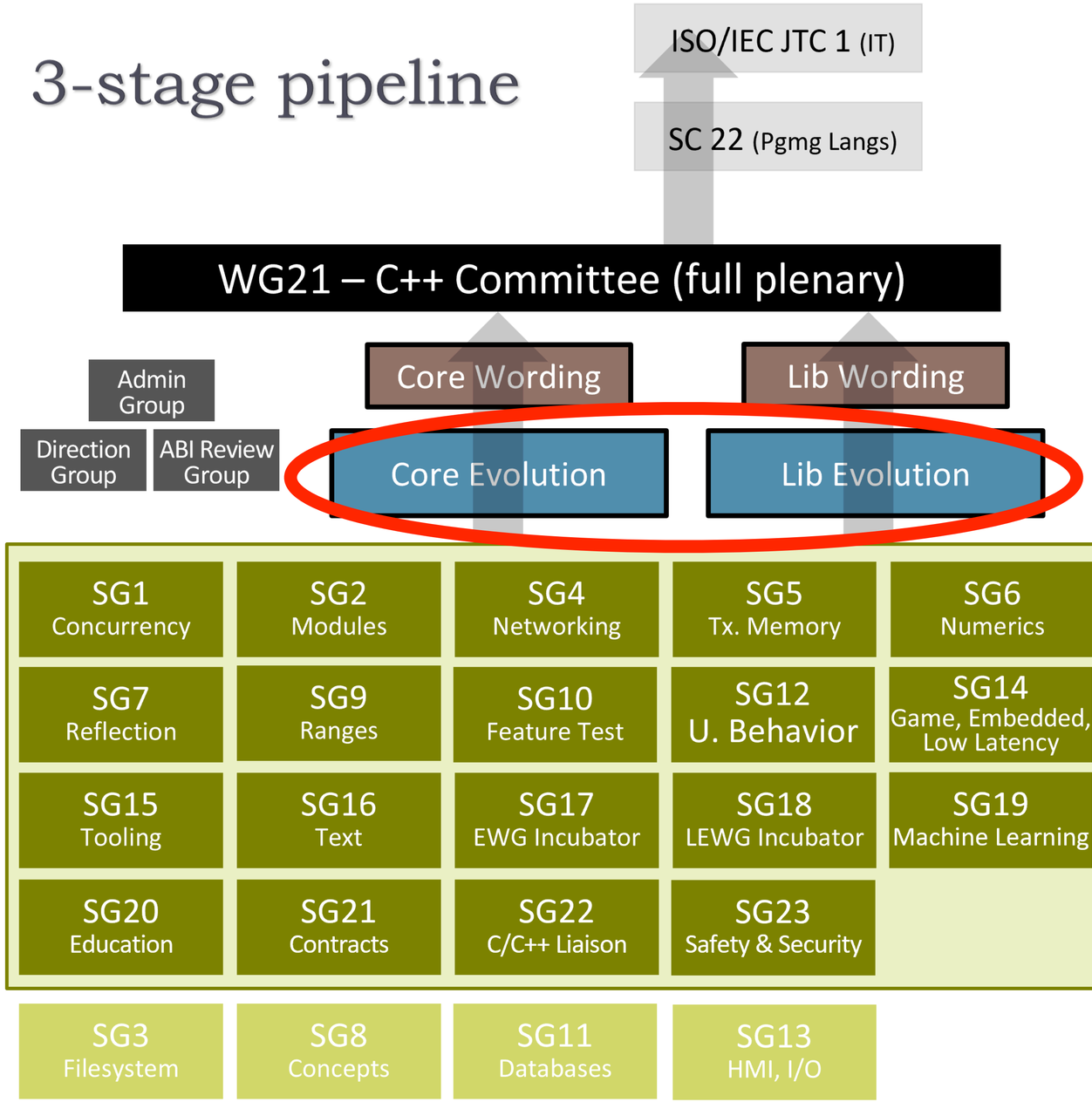
3 - Wording & Consistency

2 - Design & Target (IS/TS)

1 - Domain Specific Investigation & Incubation

Completed, inactive

3-stage pipeline



(F)DIS Approval

CD & PDTS Approval

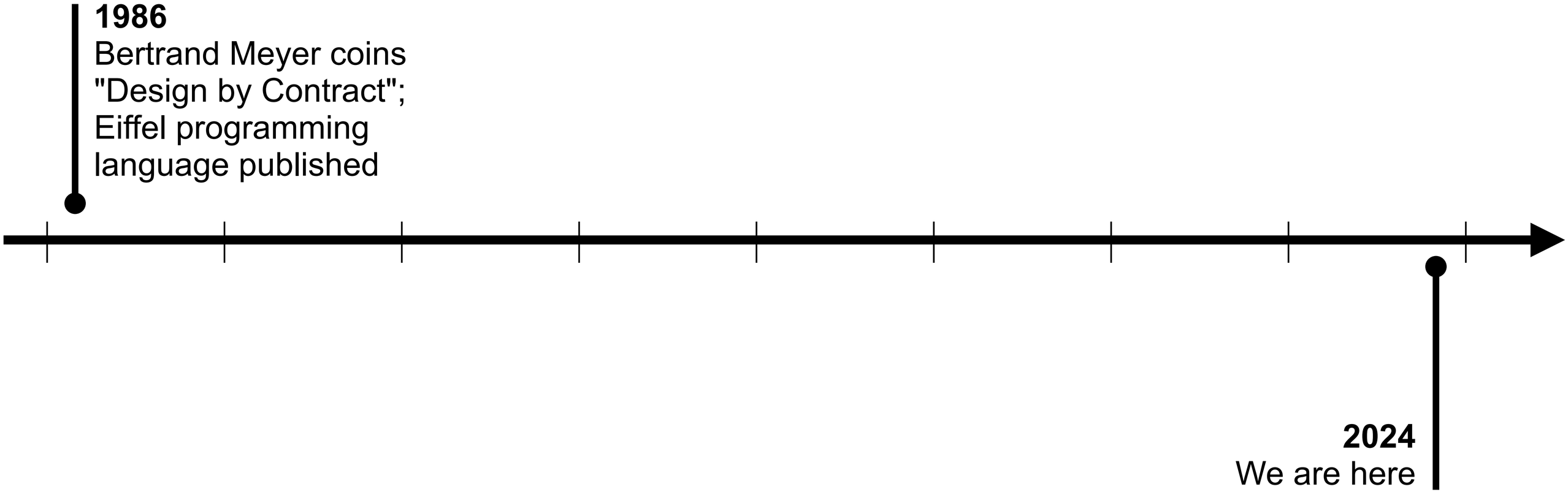
Internal Approval

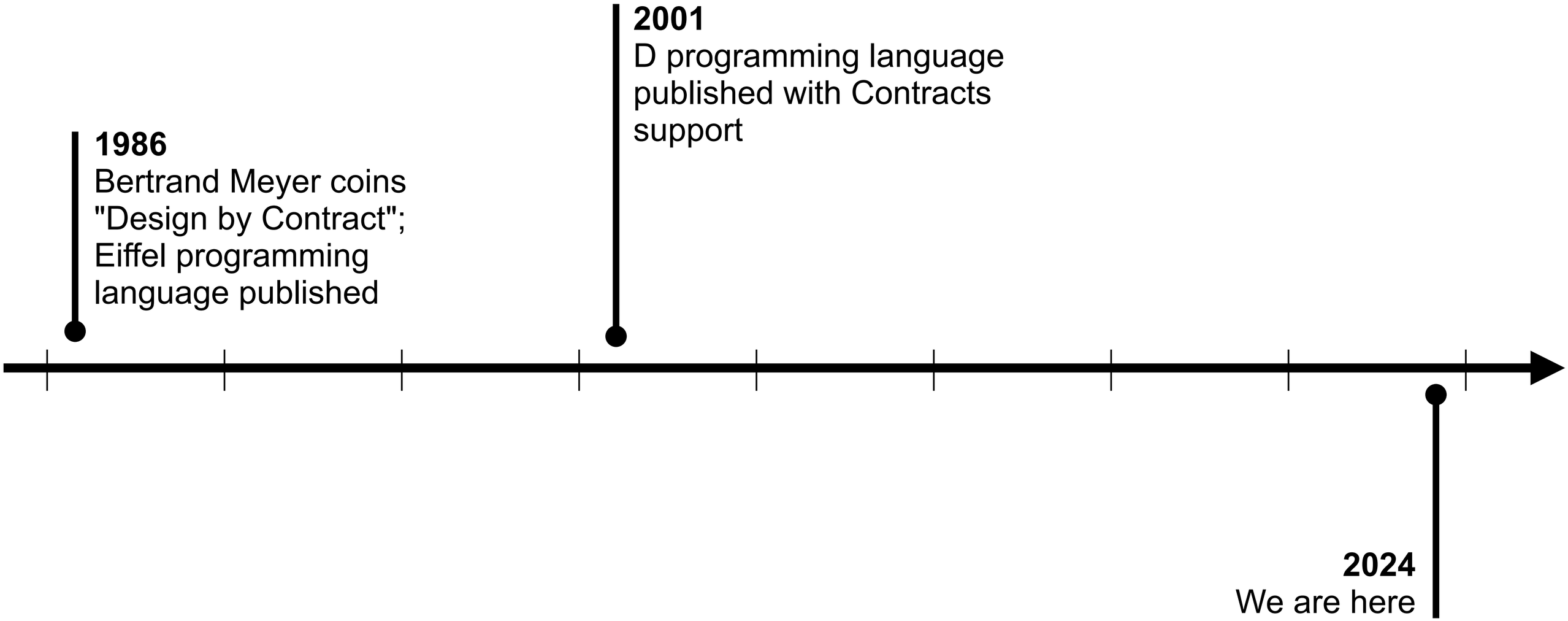
3 - Wording & Consistency

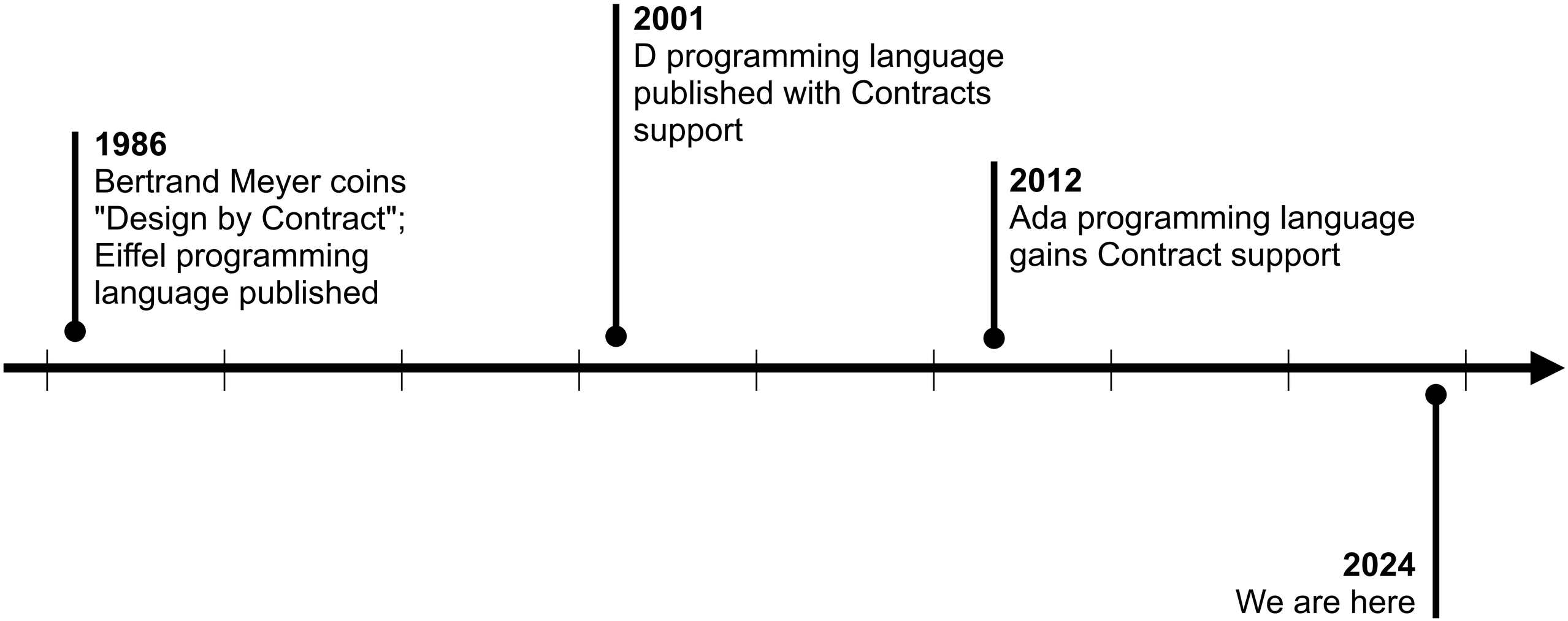
2 - Design & Target (IS/TS)

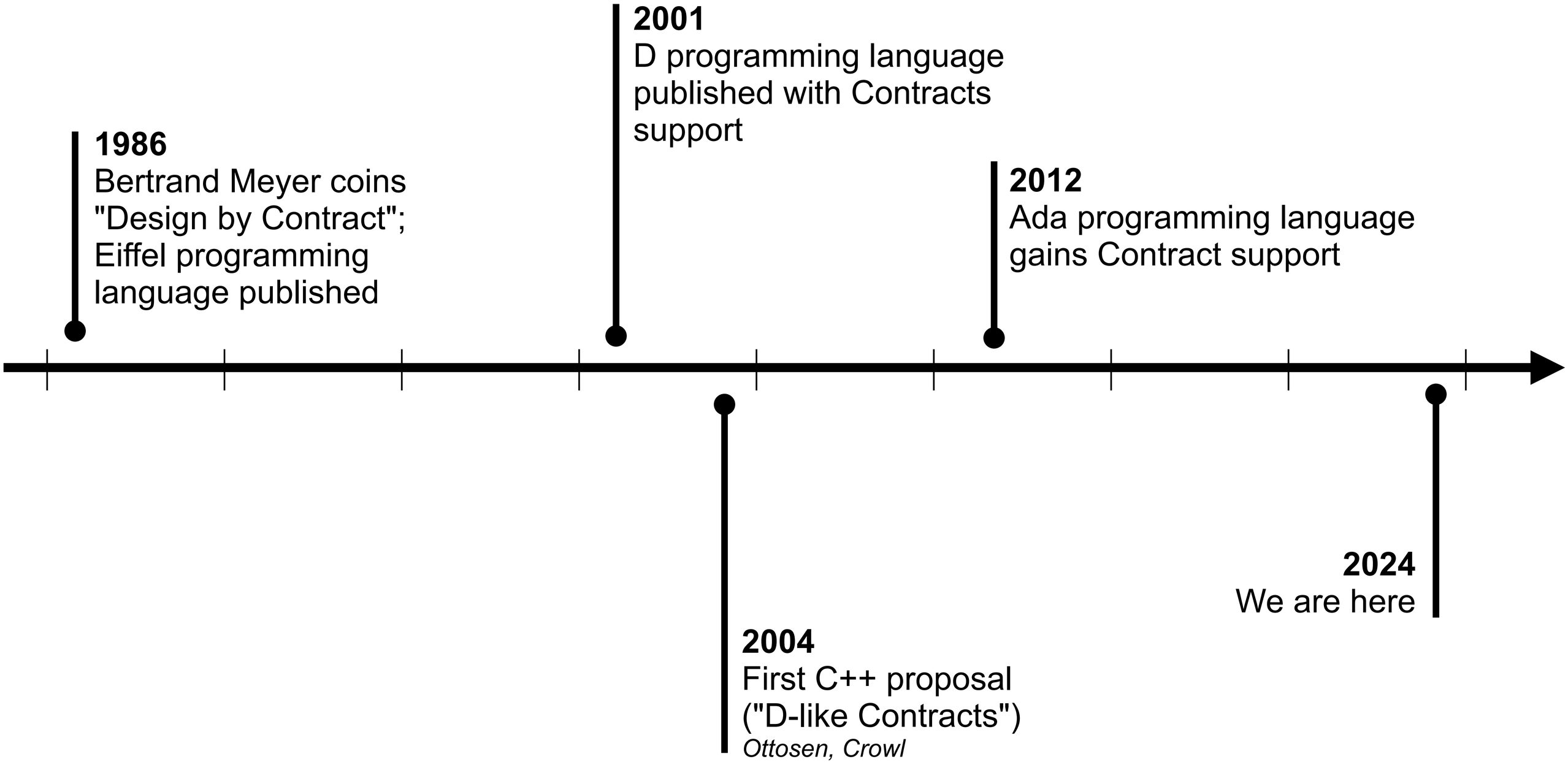
1 - Domain Specific Investigation & Incubation

Completed, inactive

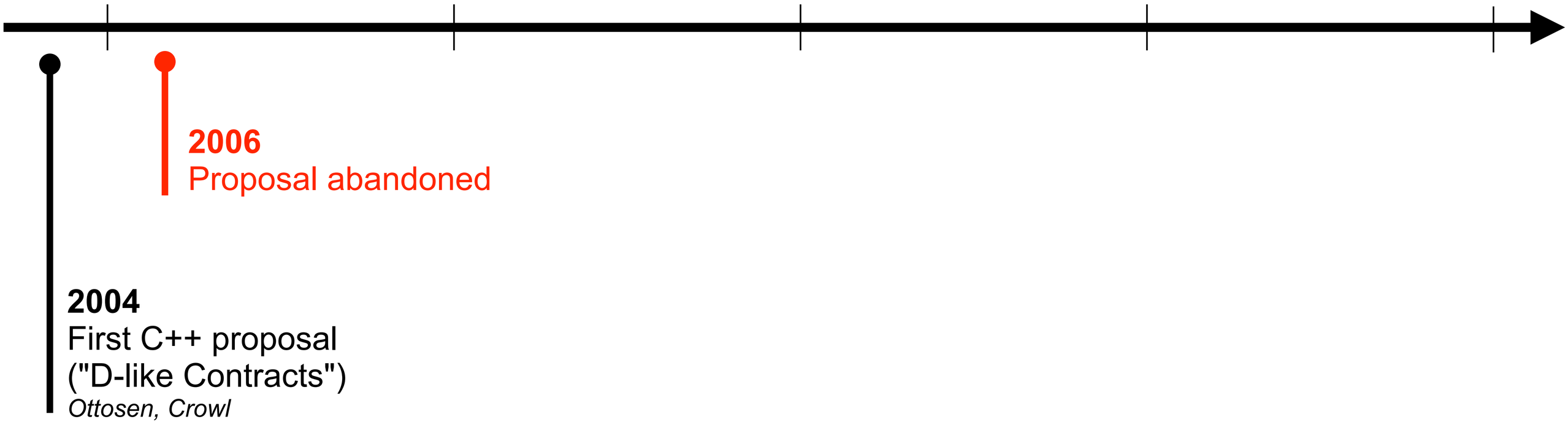

1986
Bertrand Meyer coins "Design by Contract"; Eiffel programming language published
2024
We are here

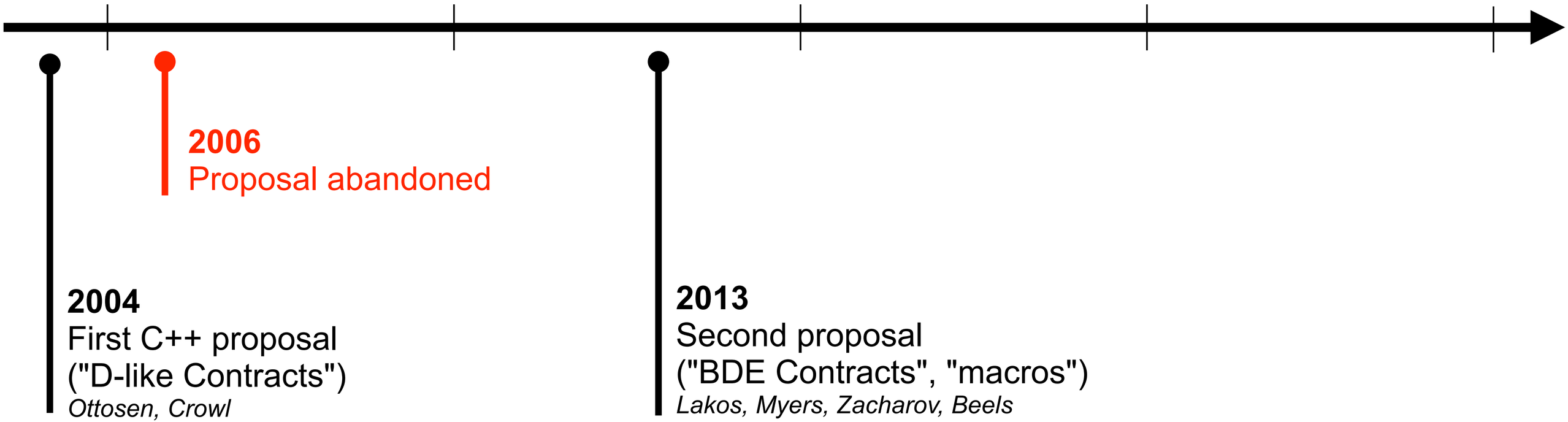

2001
D programming language published with Contracts support
1986
Bertrand Meyer coins "Design by Contract"; Eiffel programming language published
2024
We are here

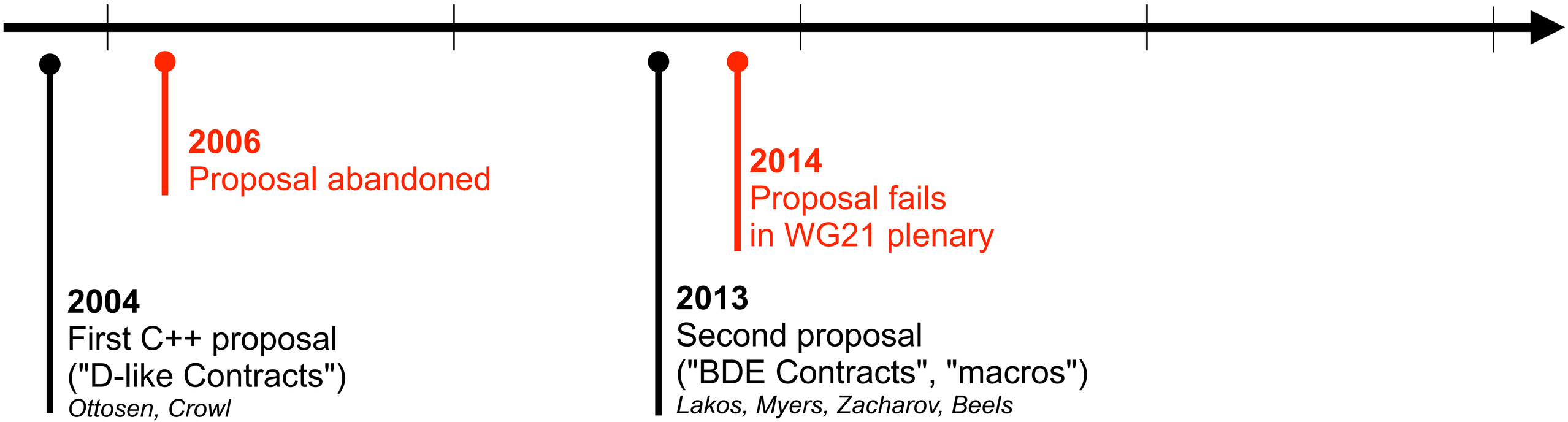

1986
Bertrand Meyer coins "Design by Contract"; Eiffel programming language published
2001
D programming language published with Contracts support
2012
Ada programming language gains Contract support
2024
We are here

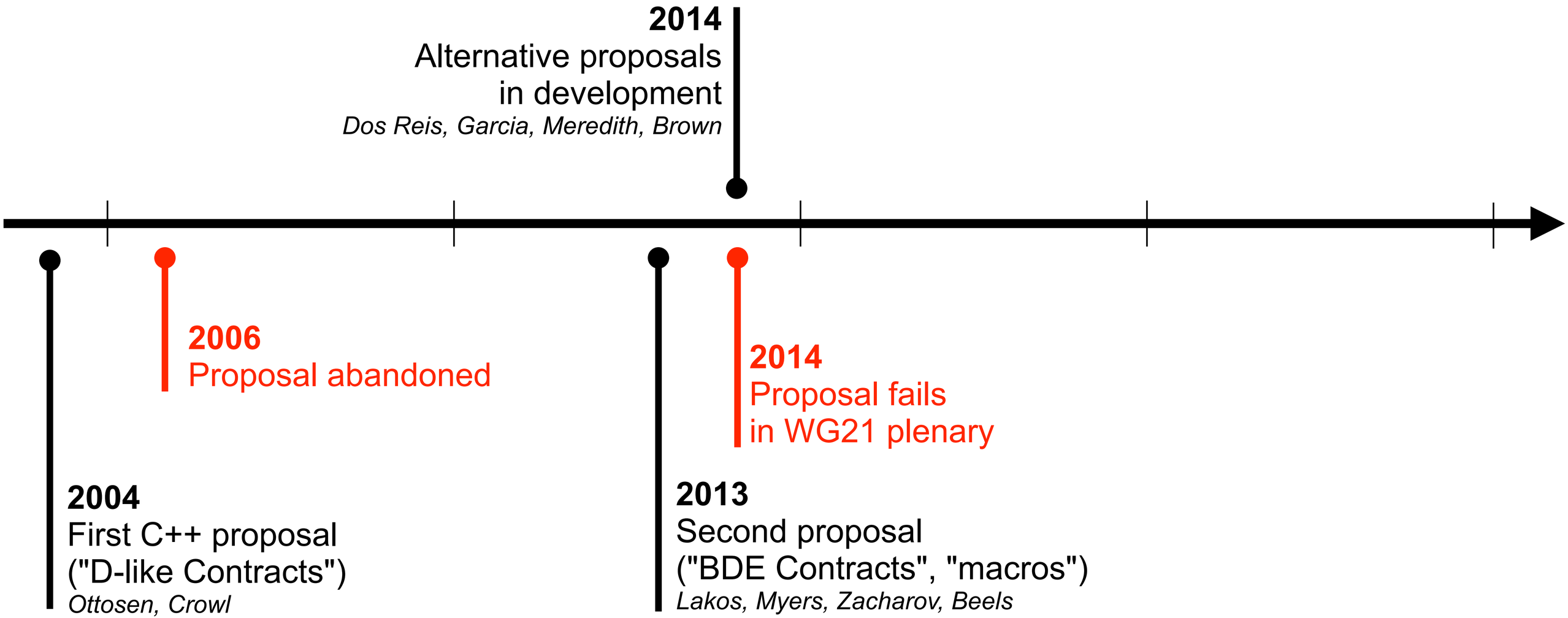

2001
D programming language published with Contracts support
1986
Bertrand Meyer coins "Design by Contract"; Eiffel programming language published
2012
Ada programming language gains Contract support
2024
We are here
2004
First C++ proposal ("D-like Contracts")
Ottosen, Crowl

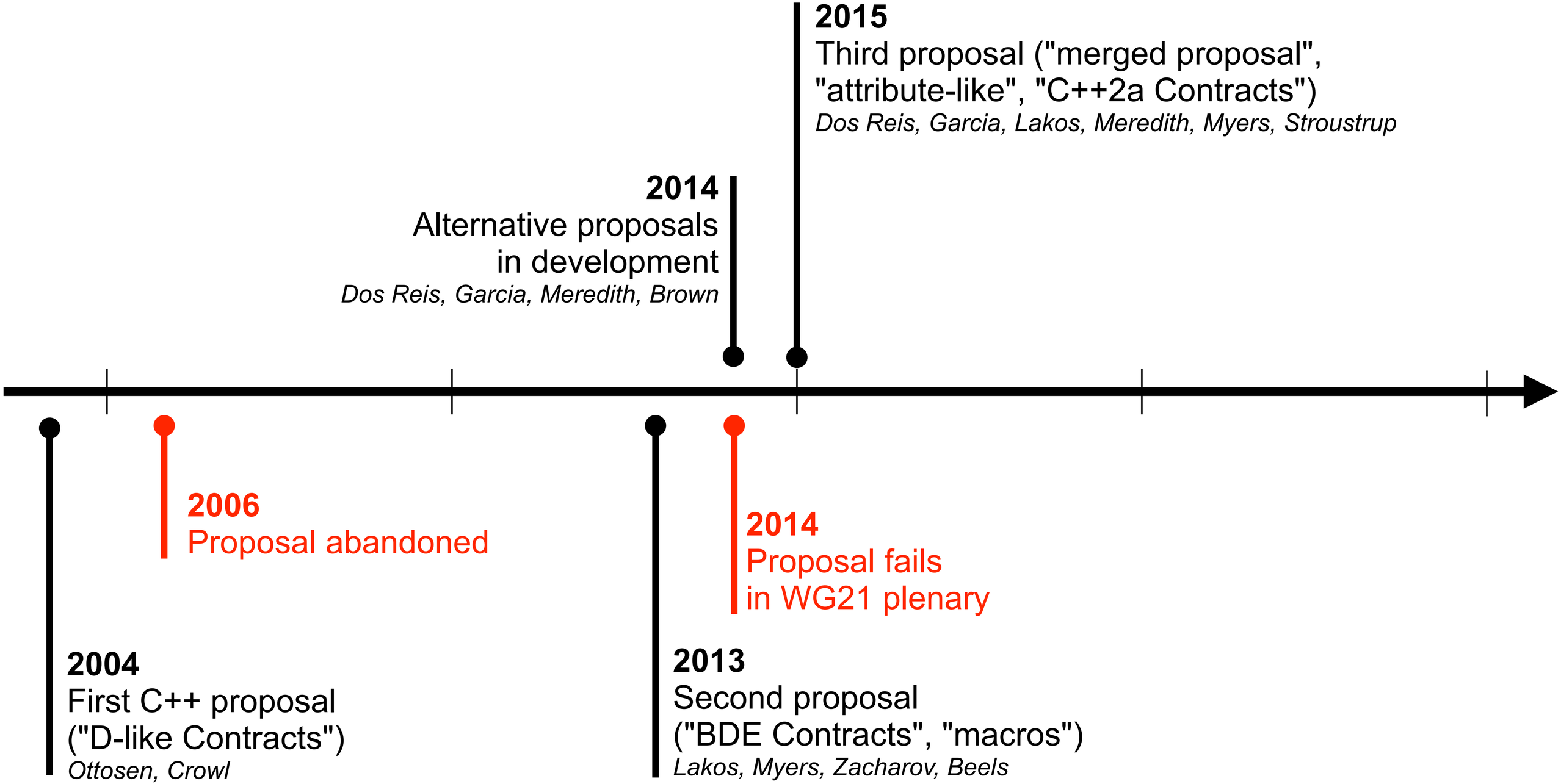
2004
First C++ proposal
("D-like Contracts")
Ottosen, Crowl

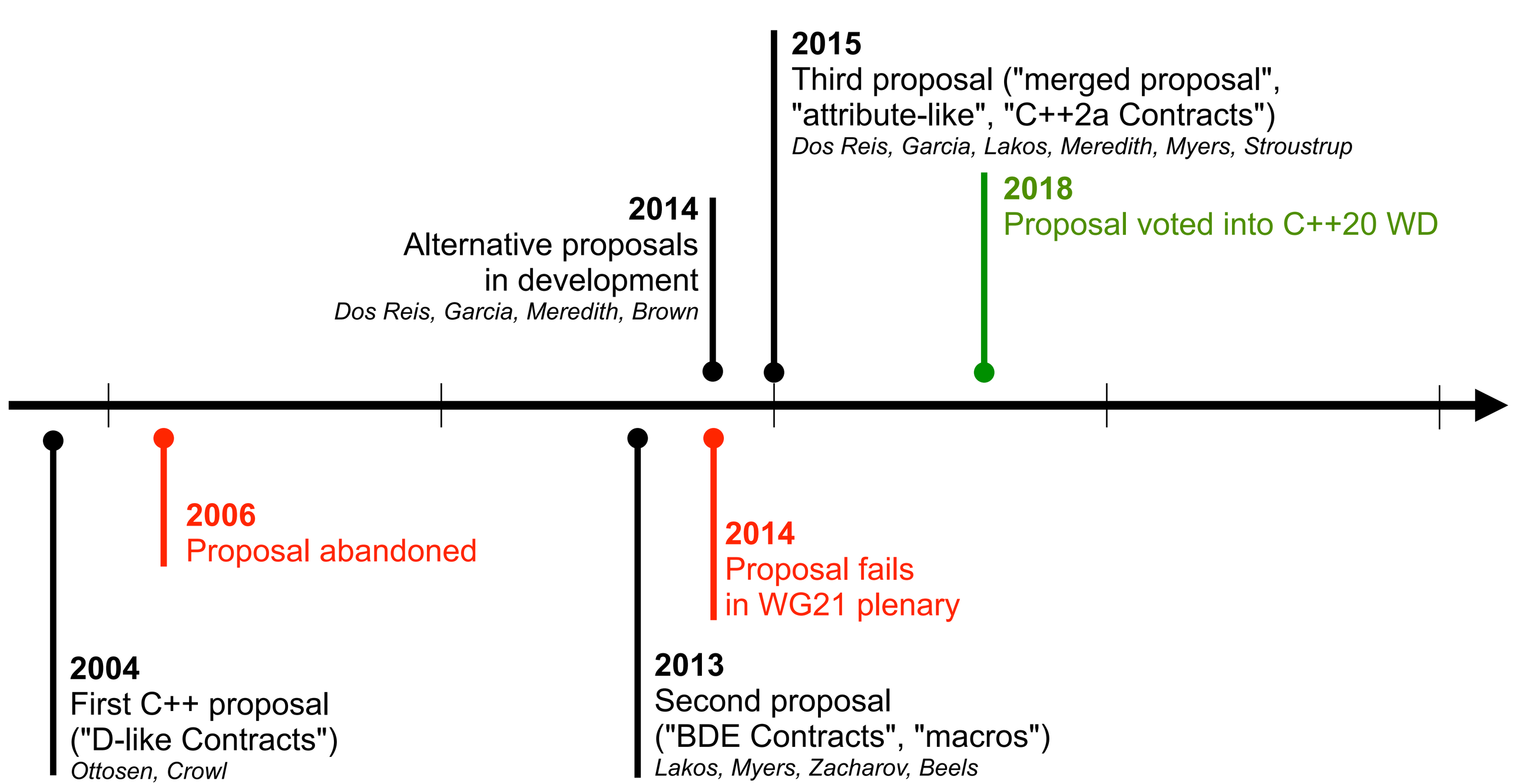

2006
Proposal abandoned
2004
First C++ proposal
("D-like Contracts")
Ottosen, Crowl

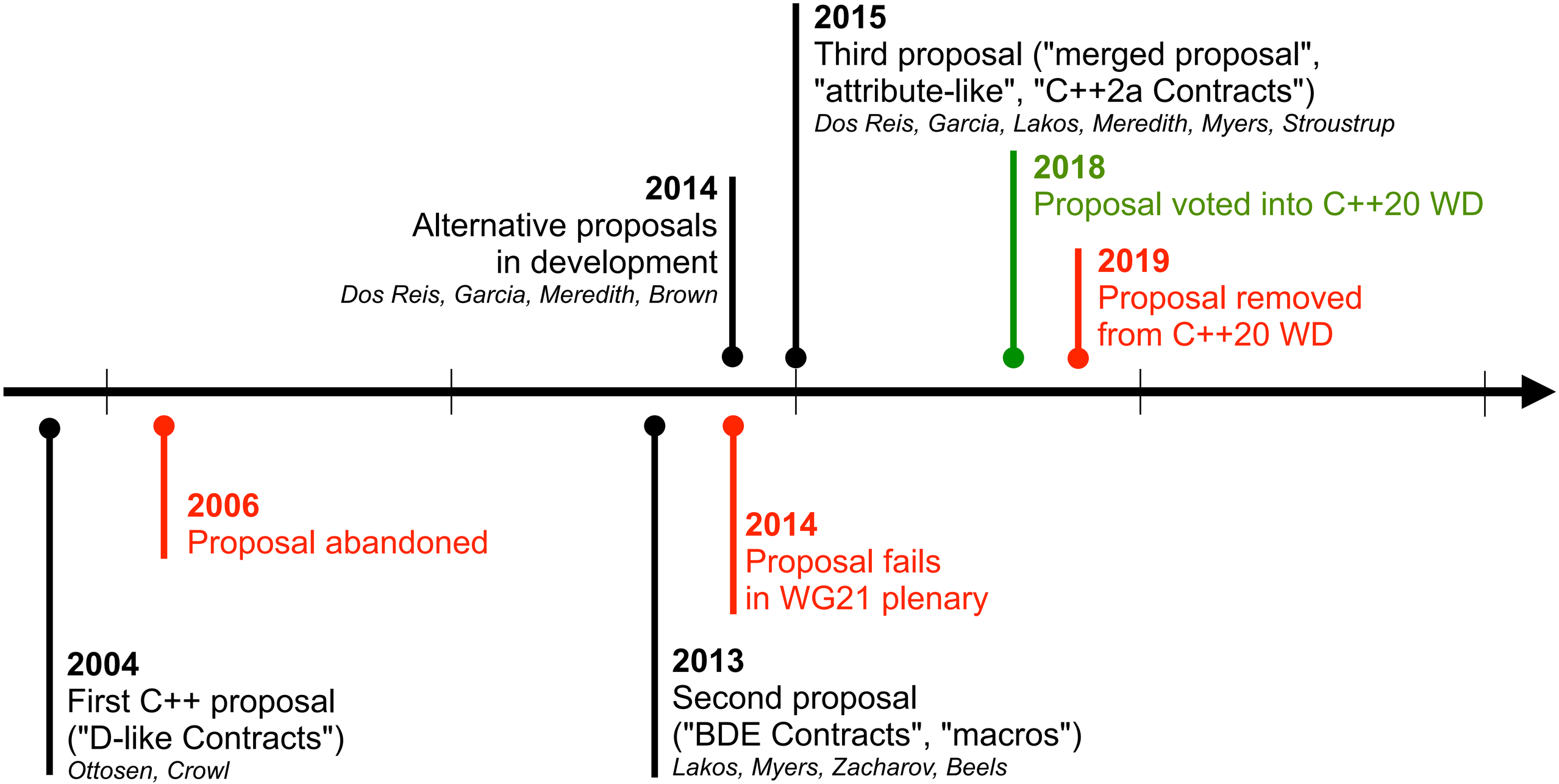

2006
Proposal abandoned
2004
First C++ proposal
("D-like Contracts")
Ottosen, Crowl
2013
Second proposal
("BDE Contracts", "macros")
Lakos, Myers, Zacharov, Beels

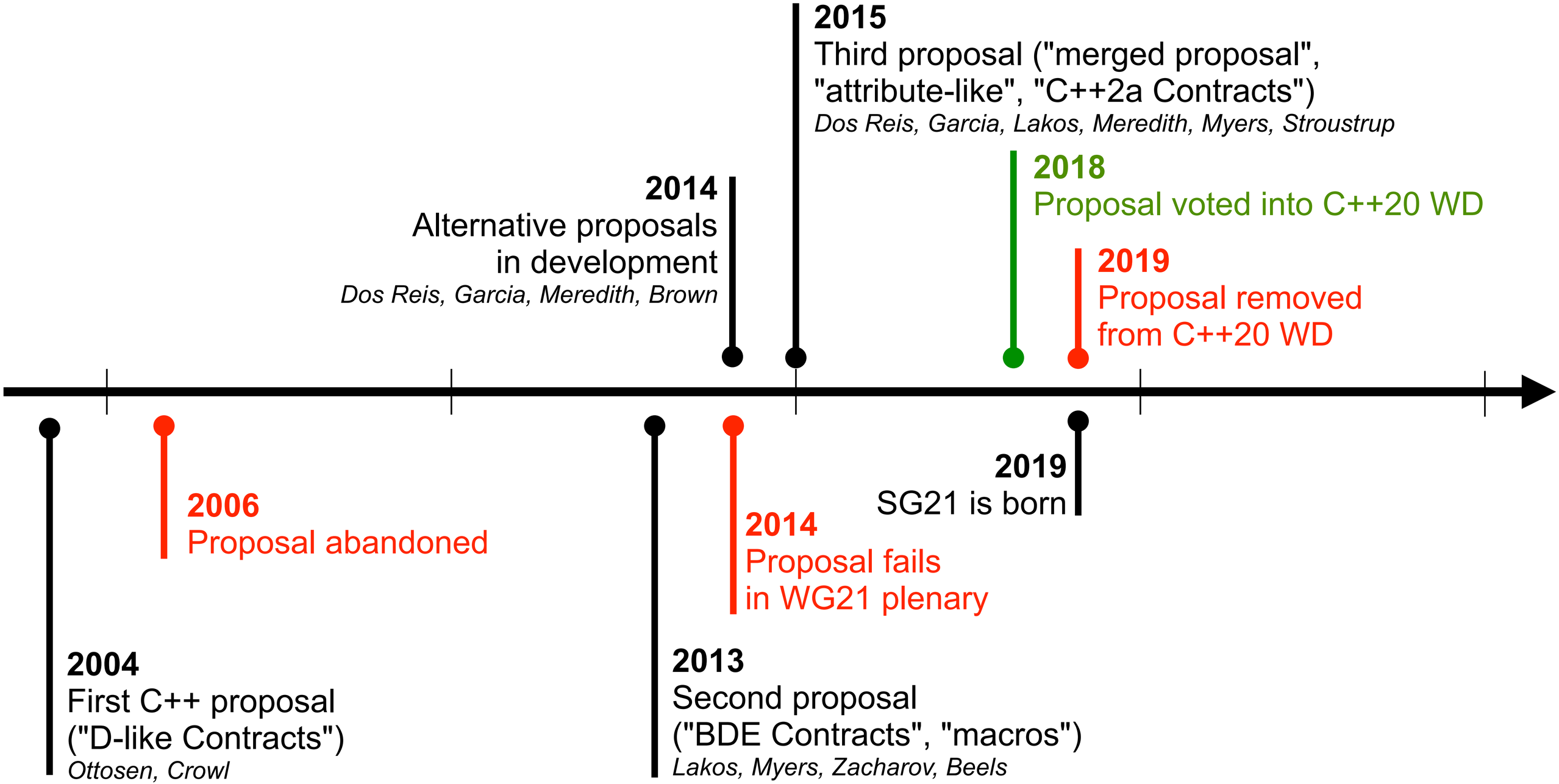

2006
Proposal abandoned
2004
First C++ proposal
("D-like Contracts")
Ottosen, Crowl
2014
Proposal fails
in WG21 plenary
2013
Second proposal
("BDE Contracts", "macros")
Lakos, Myers, Zacharov, Beels

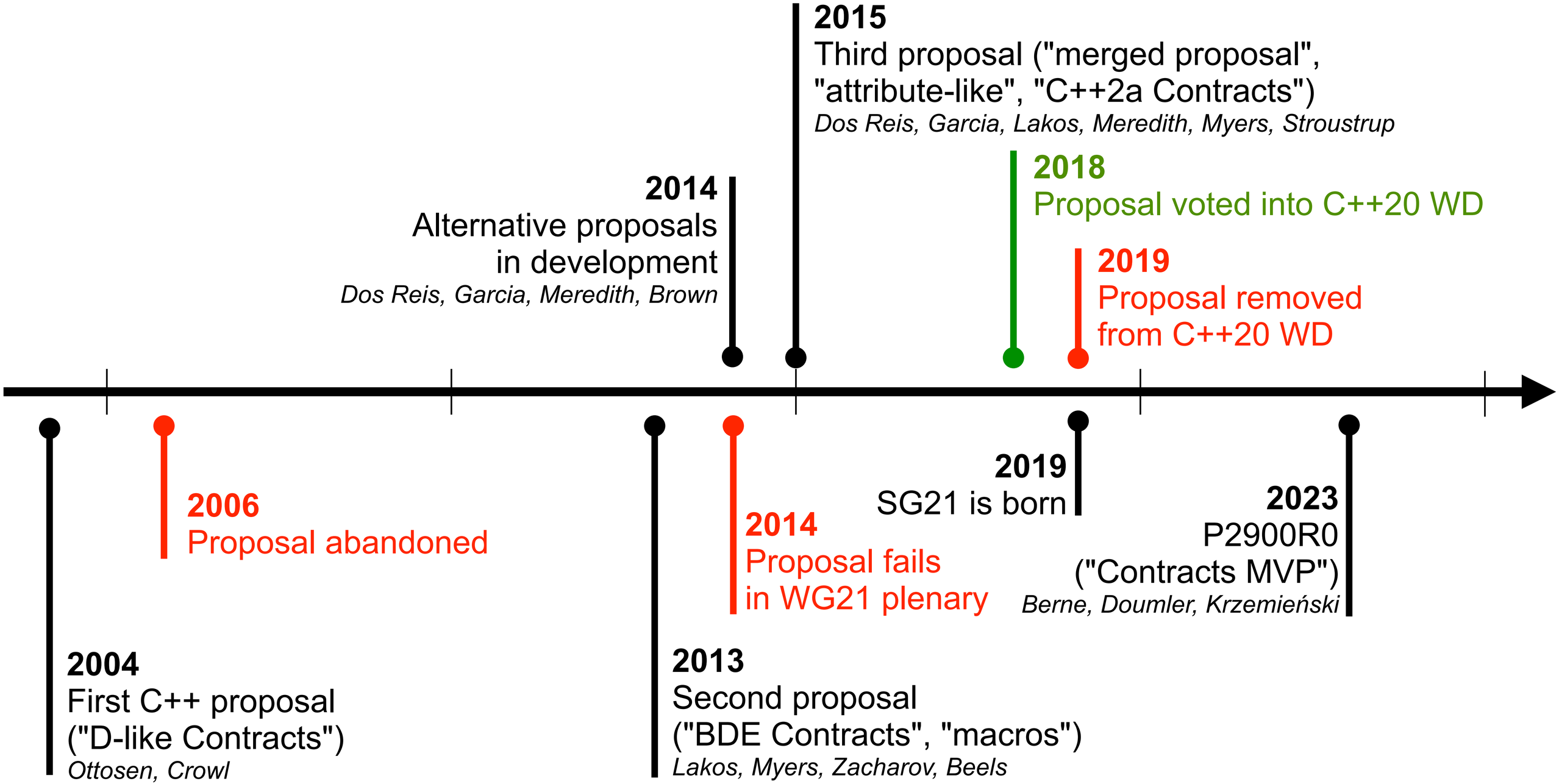

2014
Alternative proposals
in development
Dos Reis, Garcia, Meredith, Brown
2006
Proposal abandoned
2004
First C++ proposal
("D-like Contracts")
Ottosen, Crowl
2014
Proposal fails
in WG21 plenary
2013
Second proposal
("BDE Contracts", "macros")
Lakos, Myers, Zacharov, Beels

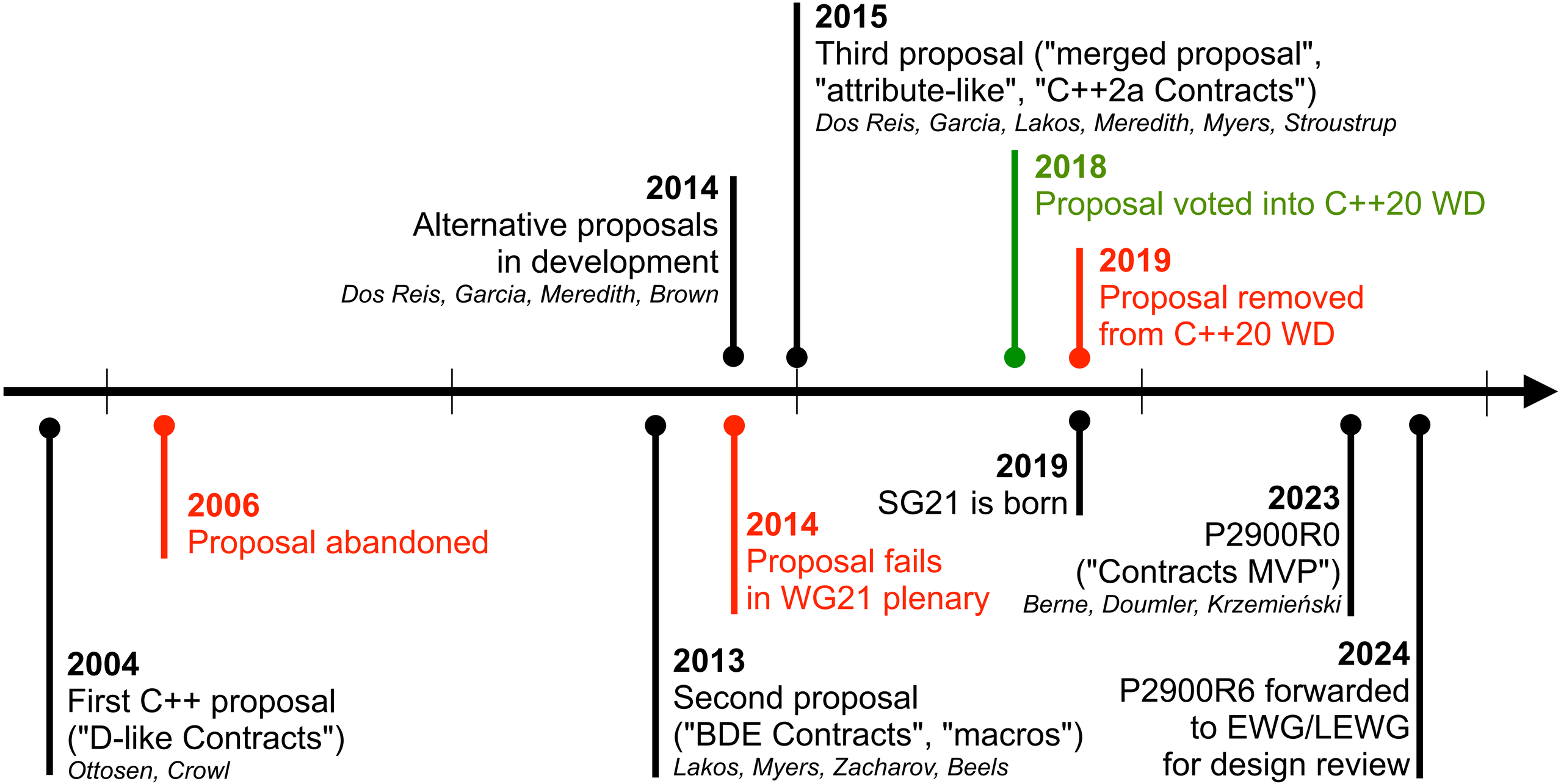

2015
Third proposal ("merged proposal", "attribute-like", "C++2a Contracts")
Dos Reis, Garcia, Lakos, Meredith, Myers, Stroustrup
2014
Alternative proposals in development
Dos Reis, Garcia, Meredith, Brown
2006
Proposal abandoned
2014
Proposal fails in WG21 plenary
2004
First C++ proposal ("D-like Contracts")
Ottosen, Crowl
2013
Second proposal ("BDE Contracts", "macros")
Lakos, Myers, Zacharov, Beels

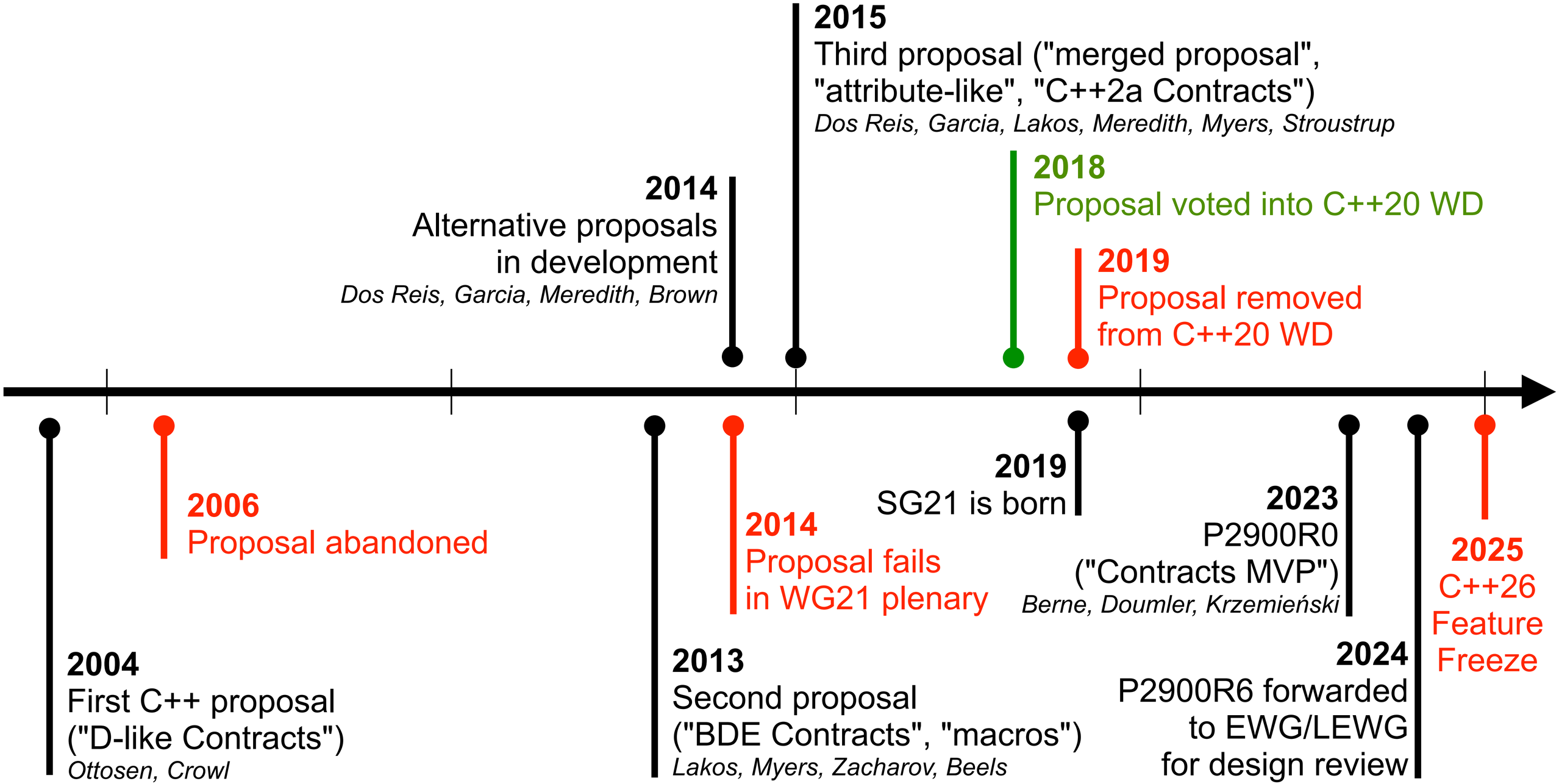

2015
Third proposal ("merged proposal", "attribute-like", "C++2a Contracts")
Dos Reis, Garcia, Lakos, Meredith, Myers, Stroustrup
2018
Proposal voted into C++20 WD
2014
Alternative proposals in development
Dos Reis, Garcia, Meredith, Brown
2006
Proposal abandoned
2014
Proposal fails in WG21 plenary
2004
First C++ proposal ("D-like Contracts")
Ottosen, Crowl
2013
Second proposal ("BDE Contracts", "macros")
Lakos, Myers, Zacharov, Beels


2015
Third proposal ("merged proposal", "attribute-like", "C++2a Contracts")
Dos Reis, Garcia, Lakos, Meredith, Myers, Stroustrup
2018
Proposal voted into C++20 WD
2014
Alternative proposals in development
Dos Reis, Garcia, Meredith, Brown
2019
Proposal removed from C++20 WD
2006
Proposal abandoned
2014
Proposal fails in WG21 plenary
2004
First C++ proposal ("D-like Contracts")
Ottosen, Crowl
2013
Second proposal ("BDE Contracts", "macros")
Lakos, Myers, Zacharov, Beels


2015
Third proposal ("merged proposal", "attribute-like", "C++2a Contracts")
Dos Reis, Garcia, Lakos, Meredith, Myers, Stroustrup
2014
Alternative proposals in development
Dos Reis, Garcia, Meredith, Brown
2018
Proposal voted into C++20 WD
2019
Proposal removed from C++20 WD
2006
Proposal abandoned
2004
First C++ proposal ("D-like Contracts")
Ottosen, Crowl
2019
SG21 is born
2014
Proposal fails in WG21 plenary
2013
Second proposal ("BDE Contracts", "macros")
Lakos, Myers, Zacharov, Beels


2015
Third proposal ("merged proposal", "attribute-like", "C++2a Contracts")
Dos Reis, Garcia, Lakos, Meredith, Myers, Stroustrup
2018
Proposal voted into C++20 WD
2014
Alternative proposals in development
Dos Reis, Garcia, Meredith, Brown
2019
Proposal removed from C++20 WD
2006
Proposal abandoned
2019
SG21 is born
2014
Proposal fails in WG21 plenary
2023
P2900R0 ("Contracts MVP")
Berne, Doumler, Krzemieński
2004
First C++ proposal ("D-like Contracts")
Ottosen, Crowl
2013
Second proposal ("BDE Contracts", "macros")
Lakos, Myers, Zacharov, Beels


2015
Third proposal ("merged proposal", "attribute-like", "C++2a Contracts")
Dos Reis, Garcia, Lakos, Meredith, Myers, Stroustrup
2018
Proposal voted into C++20 WD
2014
Alternative proposals in development
Dos Reis, Garcia, Meredith, Brown
2019
Proposal removed from C++20 WD
2006
Proposal abandoned
2019
SG21 is born
2014
Proposal fails in WG21 plenary
2023
P2900R0 ("Contracts MVP")
Berne, Doumler, Krzemieński
2004
First C++ proposal ("D-like Contracts")
Ottosen, Crowl
2013
Second proposal ("BDE Contracts", "macros")
Lakos, Myers, Zacharov, Beels
2024
P2900R6 forwarded to EWG/LEWG for design review


2015
Third proposal ("merged proposal", "attribute-like", "C++2a Contracts")
Dos Reis, Garcia, Lakos, Meredith, Myers, Stroustrup
2018
Proposal voted into C++20 WD
2014
Alternative proposals in development
Dos Reis, Garcia, Meredith, Brown
2019
Proposal removed from C++20 WD
2006
Proposal abandoned
2019
SG21 is born
2014
Proposal fails in WG21 plenary
2023
P2900R0 ("Contracts MVP")
Berne, Doumler, Krzemieński
2025
C++26 Feature Freeze
2004
First C++ proposal ("D-like Contracts")
Ottosen, Crowl
2013
Second proposal ("BDE Contracts", "macros")
Lakos, Myers, Zacharov, Beels
2024
P2900R6 forwarded to EWG/LEWG for design review

What are Contracts?

Design by Contract

- An approach for software design
- Define formal, precise and verifiable interface specifications for software components, extending their ordinary definition with:
  - preconditions
  - postconditions
  - invariants (class invariants, loop invariants...)
- called "Contracts" in accordance with a conceptual metaphor with the conditions and obligations of business contracts

Design by Contract

- An approach for software design
- Define formal, precise and verifiable interface specifications for software components, extending their ordinary definition with:
  - preconditions
  - postconditions
  - invariants (class invariants, loop invariants...)
- called "Contracts" in accordance with a conceptual metaphor with the conditions and obligations of business contracts

Design by Contract

- An approach for software design
- Define formal, precise and verifiable interface specifications for software components, extending their ordinary definition with:
  - preconditions
  - postconditions
  - invariants (class invarian
- called "Contracts" in accord ne conditions and obligations c

precondition = Condition on passed-in arguments and/or program state when a function is called. Obligation of caller (client)

postcondition = Condition on return value and/or program state when a function returns. Obligation of callee (implementer)

Contracts exist today!

Contracts exist today!

- Plain-language contracts, e.g. in code comments:

```
// Returns a reference to the element at position `index`.
// The behaviour is undefined unless `index < size()`.
T& operator[] (size_t index);
```

Contracts exist today!

- Plain-language contracts, e.g. in code comments:

```
// Returns a reference to the element at position `index`.
// The behaviour is undefined unless `index < size()`.
T& operator[] (size_t index);
```

...or in the C++ Standard:

```
constexpr const_reference operator[](size_type pos) const;
```

1 *Preconditions*: `pos < size()`.

2 *Returns*: `data_[pos]`.

3 *Throws*: Nothing.

3 Descriptions of function semantics contain the following elements (as appropriate):[143]

(3.1) — *Constraints*: the conditions for the function's participation in overload resolution ([over.match]).

[*Note 1*: Failure to meet such a condition results in the function's silent non-viability. — *end note*]

[*Example 1*: An implementation can express such a condition via a *constraint-expression* ([temp.constr.decl]). — *end example*]

(3.2) — *Mandates*: the conditions that, if not met, render the program ill-formed.

[*Example 2*: An implementation can express such a condition via the *constant-expression* in a *static_assert-declaration* ([dcl.pre]). If the diagnostic is to be emitted only after the function has been selected by overload resolution, an implementation can express such a condition via a *constraint-expression* ([temp.constr.decl]) and also define the function as deleted. — *end example*]

(3.3) — *Preconditions*: the conditions that the function assumes to hold whenever it is called; violation of any preconditions results in undefined behavior.

(3.4) — *Effects*: the actions performed by the function.

(3.5) — *Synchronization*: the synchronization operations ([intro.multithread]) applicable to the function.

(3.6) — *Postconditions*: the conditions (sometimes termed observable results) established by the function.

(3.7) — *Result*: for a *typename-specifier*, a description of the named type; for an *expression*, a description of the type of the expression; the expression is an lvalue if the type is an lvalue reference type, an xvalue if the type is an rvalue reference type, and a prvalue otherwise.

(3.8) — *Returns*: a description of the value(s) returned by the function.

(3.9) — *Throws*: any exceptions thrown by the function, and the conditions that would cause the exception.

(3.10) — *Complexity*: the time and/or space complexity of the function.

(3.11) — *Remarks*: additional semantic constraints on the function.

(3.12) — *Error conditions*: the error conditions for error codes reported by the function.

We want to express Contracts in C++ code

We want to express Contracts in C++ code

→ contract assertions

What are Contracts for?

Document no: P1995R1
Date: 2020-03-02
Authors: Joshua Berne, Timur Doumler, Andrzej Krzemieński, Ryan McDougall, Herb Sutter
Reply-to: jberne4@bloomberg.net
Audience: SG21

Contracts — Use Cases

Introduction

SG21 has gathered a large number of use cases for contracts between teh WG21 Cologne and Belfast meetings. This paper presents those use cases, along with some initial results from polling done of SG21 members to identify some level of important to the community for each individual use case.

Each use case has been assigned an identifier that can be used to reference these use cases in other papers, which will hopefully be stable. We expect this content to evolve in a number of ways:

Document no: P1995R1

Date: 2020-03-02

Authors: Joshua Berne, Timur Doumler, Andrzej Krzemieński, Ryan McDougall, Herb Sutter

Reply-to: jberne4@bloomberg.net

Audience: SG21

SG21 identified 196 use cases for Contracts

Contracts — Use Cases

Introduction

SG21 has gathered a large number of use cases for contracts between teh WG21 Cologne and Belfast meetings. This paper presents those use cases, along with some initial results from polling done of SG21 members to identify some level of important to the community for each individual use case.

Each use case has been assigned an identifier that can be used to reference these use cases in other papers, which will hopefully be stable. We expect this content to evolve in a number of ways:

performance

correctness

static analysis

optimisation

verification

debugging

formal proofs

diagnose bugs

tooling support

safety

security

annotations

documentation

runtime checks

expressivity

The Egg-Laying Wool-Milk-Pig
– German folklore

The Swan, The Pike, and The Crab
– Fable by Ivan Krylov, 1814

What are P2900 Contracts for?

P2900 Contracts enhance a C++ program
with configurable checks of its correctness,

thereby helping to diagnose and fix bugs,

across API boundaries.

– Timur Doumler

P2900 Contracts enhance a C++ program
with configurable checks of its correctness,

thereby helping to diagnose and fix bugs,

across API boundaries.

– Timur Doumler

P2900 Contracts enhance a C++ program
with configurable checks of its correctness,

thereby helping to diagnose and fix bugs,

across API boundaries.

– Timur Doumler

P2900 Contracts enhance a C++ program
with configurable checks of its correctness,

thereby helping to diagnose and fix bugs,

across API boundaries.

– Timur Doumler

P2900 Contracts enhance a C++ program
with configurable checks of its correctness,

thereby helping to diagnose and fix bugs,

across API boundaries.

– Timur Doumler

P2900 Contracts enhance a C++ program
with configurable checks of its correctness,

thereby helping to diagnose and fix bugs,

across API boundaries.

– Timur Doumler

P2900 Contracts allow the programmer to
express expectations about program state,

and optionally verify that those expectations are met.

– Lisa Lippincott

P2900 Contracts allow the programmer to
express expectations about program state,

and optionally verify that those expectations are met.

– Lisa Lippincott

P2900 Contracts allow the programmer to express expectations about program state,

and optionally verify that those expectations are met.

– Lisa Lippincott

P2900 Contracts allow the programmer to
express expectations about program state,

and optionally verify that those expectations are met.

– Lisa Lippincott

P2900 Contracts allow the programmer to
specify states that are considered incorrect
at certain points in a C++ program,

particularly when calling and returning from functions,

and then manage how such defects can be detected and mitigated,

in a portable and scalable fashion,

during program evaluation.

– Joshua Berne

P2900 Contracts allow the programmer to
specify states that are considered incorrect
at certain points in a C++ program,

particularly when calling and returning from functions,

and then manage how such defects can be detected and mitigated,

in a portable and scalable fashion,

during program evaluation.

– *Joshua Berne*

P2900 Contracts allow the programmer to
specify states that are considered incorrect
at certain points in a C++ program,

particularly when calling and returning from functions,

and then manage how such defects can be detected and mitigated,

in a portable and scalable fashion,

during program evaluation.

– *Joshua Berne*

P2900 Contracts allow the programmer to
specify states that are considered incorrect
at certain points in a C++ program,

particularly when calling and returning from functions,

and then manage how such defects can be detected and mitigated,

in a portable and scalable fashion,

during program evaluation.

– Joshua Berne

P2900 Contracts allow the programmer to
specify states that are considered incorrect
at certain points in a C++ program,

particularly when calling and returning from functions,

and then manage how such defects can be detected and mitigated,

in a portable and scalable fashion,

during program evaluation.

– *Joshua Berne*

P2900 Contracts allow the programmer to
specify states that are considered incorrect
at certain points in a C++ program,

particularly when calling and returning from functions,

and then manage how such defects can be detected and mitigated,

in a portable and scalable fashion,

during program evaluation.

– Joshua Berne

```
// Returns a reference to the element at position `index`.
// The behaviour is undefined unless `index < size()`.
T& operator[] (size_t index);
```

```
  // Returns a reference to the element at position `index`.
  // The behaviour is undefined unless `index < size()`.
  T& operator[] (size_t index) {
    return _data[index];
  }

private:
  T* _data;
```

```
  // Returns a reference to the element at position `index`.
  // The behaviour is undefined unless `index < size()`.
  T& operator[] (size_t index) {
    return _data[index]; // potentially UB here :(
  }

private:
  T* _data;
```

```
T& at (size_t index) {
  if (index >= size())
    throw std::logic_error("Index out of bounds!");

  return _data[index];
}
```

```
T& safe_get_element (size_t index) {
  if (index >= size())
    std::terminate();

  return _data[index];
}
```

```
T& operator[] (size_t index)
  pre (index < size());
```

Caller / client

reads contract assertions

ensures preconditions are met

expects postconditions to be met

Callee / implementer

writes contract assertions

expects preconditions to be met

ensures postconditions are met

Caller / client

reads contract assertions

ensures preconditions are met

expects postconditions to be met

Callee / implementer

writes contract assertions

expects preconditions to be met

ensures postconditions are met

Build engineer / owner of main()

enables & configures contract checks

decides what happens in case of contract violation

```
#include <cassert>

T& operator[] (size_t index) {
  assert (index < size());
  return _data[index];
}
```

cassert is not enough

What are P2900 Contracts for?

P2900 Contracts provide a superior replacement for cassert and similar macros.

P2900 Contracts are not macros

```
#include <cassert>
assert(Widget{1, 2}.isValid());
                   ^~~~~~~~~~~
                   parse error

assert(array<int, 3>{}.size() == 3);
                    ^~~~~~~~~~~
                    parse error
```

P2900 Contracts are not macros

```
#include <cassert>
assert(Widget{1, 2}.isValid());
                   ^~~~~~~~~~~
                   parse error

assert(array<int, 3>{}.size() == 3);
                    ^~~~~~~~~~~
                    parse error
```

```
contract_assert(Widget{1, 2}.isValid());
// OK

contract_assert(array<int, 3>{}.size() == 3);
// OK
```

P2900 Contracts are not macros

```
// 1.cpp
void f(int i) {
  assert(i >= 0);
  // ...
}

// 2.cpp -DNDEBUG
void f(int i) {
  assert(i >= 0);
  // ...
}

// ODR violation :(
```

P2900 Contracts are not macros

```
// 1.cpp
void f(int i) {
  assert(i >= 0);
  // ...
}

// 2.cpp -DNDEBUG
void f(int i) {
  assert(i >= 0);
  // ...
}

// ODR violation :(
```

```
// 1.cpp -fcontract-semantics=enforce
void f(int i)
pre (i >= 0) {
  // ...
}

// 2.cpp -fcontract-semantics=ignore
void f(int i)
pre (i >= 0) {
  // ...
}

// OK
```

P2900 Contracts are not macros

```
// 1.cpp -DNDEBUG
void g() {
  assert(doesNotExist); // compiles
  // ...
}
```

P2900 Contracts are not macros

```
// 1.cpp -DNDEBUG
void g() {
  assert(doesNotExist); // compiles
  // ...
}

// ...until you turn off NDEBUG :(
```

P2900 Contracts are not macros

```
// 1.cpp -DNDEBUG
void g() {
  assert(doesNotExist); // compiles
  // ...
}

// ...until you turn off NDEBUG :(
```

```
// 1.cpp -fcontract-semantics=ignore
void g() {
  contract_assert(doesNotExist);
  // ...          ~~~~~~~~~~~~~~~~~~
}                 parsed as C++ code
                  even if contract checks
                  are disabled!
```

P2900 `pre` / `post` can go on declarations

```
// vector.h
T& operator[] (size_t i);
// can't put assert macro here :(

void resize(size_t n);
// can't put assert macro here :(
```

P2900 `pre` / `post` can go on declarations

```
// vector.h
T& operator[] (size_t i);
// can't put assert macro here :(

void resize(size_t n);
// can't put assert macro here :(
```

```
// vector.h
T& operator[] (size_t i)
  pre (i < size());

void resize(size_t n)
  post (size() == n);
```

P2900 `pre` / `post` can go on declarations

```
// vector.h
T& operator[] (size_t i);
// can't put assert macro here :(

void resize(size_t n);
// can't put assert macro here :(
```

```
// vector.h
T& operator[] (size_t i)
  pre (i < size());

void resize(size_t n)
  post (size() == n);

// implementation:
T& operator[] (size_t i)
pre (i < size())  // optional here!
{
  return _data[i];
}
```

P2900 `post` can directly refer to return value

```
Widget getWidget()
  post (r : r.isValid());  // `r` names the returned object
```

Works with guaranteed copy elision!

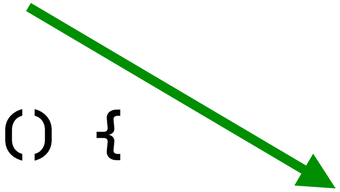
```
std::atomic<int> f()
  post(ret: ret == 0);

int main() {
  std::atomic<int> a = f();  // a is non-copyable & non-movable
}
```

Works with guaranteed copy elision!

```
std::atomic<int> f()
  post(ret: ret == 0);        // `ret` refers to `a` in main()!

int main() {
  std::atomic<int> a = f();  // a is non-copyable & non-movable
}
```



Point of evaluation

- **Precondition assertions (pre):**
  after the initialisation of function parameters,
  before the evaluation of the function body
- **Postcondition assertions (post):**
  after the result object value has been initialised
  and local automatic variables have been destroyed,
  but prior to the destruction of function parameters
- **Assertion statements (contract_assert):**
  when the statement is executed

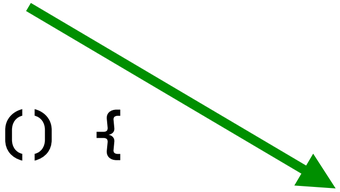
Point of evaluation

- **Precondition assertions (pre):**
  after the initialisation of function parameters,
  before the evaluation of the function body
- **Postcondition assertions (post):**
  after the result object value has been initialised
  and local automatic variables have been destroyed,
  but prior to the destruction of function parameters
- **Assertion statements (contract_assert):**
  when the statement is executed

Works with guaranteed copy elision!

```
std::atomic<int> f()
  post(ret: ret == 0);        // `ret` refers to `a` in main()!

int main() {
  std::atomic<int> a = f();  // a is non-copyable & non-movable
}
```



`pre` / `post` can refer to private variables

```
struct X {
  void f(int j)
    pre (j != i);  // OK; name lookup as-if first statement in body

private:
  int i = 0;
};
```

Arbitrary number of `pre` / `post` assertions

```
int clamp(const int val, const int min, const int max)
  pre (min <= max)
  post (r: val < min ? r == min : r == val)
  post (r: val > max ? r == max : r == val);
```

Referring to non-reference parameters in `post`

```
int clamp(const int val, const int min, const int max)
  pre (min <= max)
  post (r: val < min ? r == min : r == val)
  post (r: val > max ? r == max : r == val);
```

Referring to non-reference parameters in `post`

```
int clamp(const int val, const int min, const int max)
  pre (min <= max)
  post (r: val < min ? r == min : r == val)    // parameters must be const
  post (r: val > max ? r == max : r == val);   // on all declarations!
```

Referring to non-reference parameters in `post`

```
int clamp(int v, int min, int max)
  pre (min <= max);
  post (r: val < min ? r == min : r == val)
  post (r: val > max ? r == max : r == val)
{
  min = max = value = 0;
  return 0;
}

// this won't compile!
```

What happens when contract check fails?

What happens when contract check fails?

```
#include <cassert>
assert(false);
// logs a message to stdout & terminates
```

What happens when contract check fails?

```
#include <cassert>
assert(false);
// logs a message to stdout & terminates
```

```
contract_assert(false);
// calls the contract-violation handler
```

What happens when contract check fails?

```
#include <cassert>
assert(false);
// logs a message to stdout & terminates
```

```
contract_assert(false);
// calls the contract-violation handler
// -> default handler:
//    logs a message to stdout & terminates
```

Customising the contract-violation handler

- cassert: impossible
- custom assertion macros: possible, but not across components
- P2900 contracts: possible, you can define one handler for the entire program at link time!
  (replaceable function / weak symbol, like **`operator new`**)

User-defined contract-violation handler

```
void ::handle_contract_violation
(const std::contracts::contract_violation& violation)
{
  LOG(std::format("Contract violated at: {}\n", violation.location()));
}
```

User-defined contract-violation handler

```
void ::handle_contract_violation
(const std::contracts::contract_violation& violation)
{
  std::breakpoint();
}
```

User-defined contract-violation handler

```
void ::handle_contract_violation
(const std::contracts::contract_violation& violation)
{
  while (!std::is_debugger_present())
    /* spin */;

  std::breakpoint();
}
```

User-defined contract-violation handler

```
void ::handle_contract_violation
(const std::contracts::contract_violation& violation)
{
  std::cout << std::stacktrace::current(1);
}
```

User-defined contract-violation handler

```
void ::handle_contract_violation
(const std::contracts::contract_violation& violation)
{
  std::cout << std::stacktrace::current(1);
  std::contracts::invoke_default_contract_violation_handler(violation);
}
```

User-defined contract-violation handler

```
void ::handle_contract_violation
(const std::contracts::contract_violation& violation)
{
  throw my::contract_violation_exception(violation);
}
```

User-defined contract-violation handler

```
void ::handle_contract_violation
(const std::contracts::contract_violation& violation)
{
  throw my::contract_violation_exception(violation);
}
```

Lakos Rule = Do not put noexcept on a function with preconditions, even if it never throws when called correctly!

C++ Standard follows the Lakos Rule, e.g. **`std::vector::operator[]`**

Standard Library API

```
namespace std::contracts {
  class contract_violation {
    // No user-accessible constructor, not copyable/movable/assignable
  public:
    std::source_location location() const noexcept;
    const char* comment() const noexcept;
    detection_mode detection_mode() const noexcept;
    evaluation_semantic semantic() const noexcept;
    assertion_kind kind() const noexcept;
  };
  void invoke_default_contract_violation_handler(const contract_violation&);
}
```

Standard Library API

```
namespace std::contracts {
  class contract_violation {
    // No user-accessible constructor, not copyable/movable/assignable
  public:
    std::source_location location() const noexcept;
    const char* comment() const noexcept;
    detection_mode detection_mode() const noexcept;
    evaluation_semantic semantic() const noexcept;
    assertion_kind kind() const noexcept;
  };
  void invoke_default_contract_violation_handler(const contract_violation&);
}
```

Standard Library API

```
namespace std::contracts {
  class contract_violation {
    // No user-accessible constructor, not copyable/movable/assignable
  public:
    std::source_location location() const noexcept;
    const char* comment() const noexcept;
    detection_mode detection_mode() const noexcept;
    evaluation_semantic semantic() const noexcept;
    assertion_kind kind() const noexcept;
  };
  void invoke_default_contract_violation_handler(const contract_violation&);
}
```

Standard Library API

```
namespace std::contracts {
  class contract_violation {
    // No user-accessible constructor, not copyable/movable/assignable
  public:
    std::source_location location() const noexcept;
    const char* comment() const noexcept;
    detection_mode detection_mode() const noexcept;
    evaluation_semantic semantic() const noexcept;
    assertion_kind kind() const noexcept;
  };
  void invoke_default_contract_violation_handler(const contract_violation&);
}
```

```
namespace std::contracts {
  enum class detection_mode : int {
    predicate_false = 1,
    evaluation_exception = 2,
    // implementation-defined additional values allowed, must be >= 1000
  };
}
```

Standard Library API

```
namespace std::contracts {
  class contract_violation {
    // No user-accessible constructor, not copyable/movable/assignable
  public:
    std::source_location location() const noexcept;
    const char* comment() const noexcept;
    detection_mode detection_mode() const noexcept;
    evaluation_semantic semantic() const noexcept;
    assertion_kind kind() const noexcept;
  };
  void invoke_default_contract_violation_handler(const contract_violation&);
}
```

```
namespace std::contracts {
  enum class evaluation_semantic : int {
    enforce = 1,
    observe = 2,
    // implementation-defined additional values allowed, must be >= 1000
  };
}
```

Standard Library API

```
namespace std::contracts {
  class contract_violation {
    No user-accessible constructor, not copyable/movable/assignable
  public:
    std::source_location location() const noexcept;
    const char* comment() const noexcept;
    detection_mode detection_mode() const noexcept;
    evaluation_semantic semantic() const noexcept;
    assertion_kind kind() const noexcept;
  };
  void invoke_default_contract_violation_handler(const contract_violation&);
}
```

```
namespace std::contracts {
  enum class assertion_kind : int {
    pre = 1,
    post = 2,
    assert = 3,
    // implementation-defined additional values allowed, must be >= 1000
  };
}
```

Standard Library API

```
namespace std::contracts {
  class contract_violation {
    No user-accessible constructor, not copyable/movable/assignable
  public:
    std::source_location location() const noexcept;
    const char* comment() const noexcept;
    detection_mode detection_mode() const noexcept;
    evaluation_semantic semantic() const noexcept;
    assertion_kind kind() const noexcept;
  };
  void invoke_default_contract_violation_handler(const contract_violation&);
}
```

Standard Library API

```
namespace std::contracts {
  class contract_violation {
    // No user-accessible constructor, not copyable/movable/assignable
  public:
    std::source_location location() const noexcept;
    const char* comment() const noexcept;
    detection_mode detection_mode() const noexcept;
    evaluation_semantic semantic() const noexcept;
    assertion_kind kind() const noexcept;
  };
  void invoke_default_contract_violation_handler(const contract_violation&);
}
```

Standard Library API

- Everything is in namespace **`std::contracts`**
- Everything is in header **`<contracts>`**
  - Only needed to implement a user-defined contract-violation handler, not needed to add **`pre`** / **`post`** / **`contract_assert`** to your code!

Caller / client

reads contract assertions

ensures preconditions are met

expects postconditions to be met

Callee / implementer

writes contract assertions

expects preconditions to be met

ensures postconditions are met

Build engineer / owner of main()

enables & configures contract checks

decides what happens in case of contract violation

Caller / client

reads contract assertions

ensures preconditions are met

expects postconditions to be met

Callee / implementer

writes contract assertions

expects preconditions to be met

ensures postconditions are met

Build engineer / owner of main()

enables & configures contract checks

decides what happens in case of contract violation

How?

A contract assertion can be evaluated with one of the following evaluation semantics:

- ***ignore***: do not check the predicate (but still parse it)

A contract assertion can be evaluated with one of the following evaluation semantics:

- ***ignore***: do not check the predicate (but still parse it)
- ***observe***: check the predicate, if the check fails call the contract-violation handler, when handler returns continue

A contract assertion can be evaluated with one of the following evaluation semantics:

- ***ignore***: do not check the predicate (but still parse it)
- ***observe***: check the predicate, if the check fails call the contract-violation handler, when handler returns continue
- ***enforce***: check the predicate, if the check fails call the contract-violation handler, when handler returns terminate the program

A contract assertion can be evaluated with one of the following evaluation semantics:

- ***ignore***: do not check the predicate (but still parse it)
- ***observe***: check the predicate, if the check fails call the contract-violation handler, when handler returns continue
- ***enforce***: check the predicate, if the check fails call the contract-violation handler, when handler returns terminate the program
- ***quick_enforce***: check the predicate, if the check fails immediately terminate the program

A contract assertion can be evaluated with one of the following evaluation semantics:

- ***assume***: do not check the predicate and optimise on the assumption that it is `true` (= if it is `false`, the behaviour is undefined) - not in P2900
- ***ignore***: do not check the predicate (but still parse it)
- ***observe***: check the predicate, if the check fails call the contract-violation handler, when handler returns continue
- ***enforce***: check the predicate, if the check fails call the contract-violation handler, when handler returns terminate the program
- ***quick_enforce***: check the predicate, if the check fails immediately terminate the program

Choice of semantic is implementation-defined

- Can be chosen at compile time, e.g. `-fcontract-semantic=enforce`
- Can be chosen at load time
- Can be chosen at link time
- Can be chosen at runtime
  - e.g. enable assertions when a debugger is attached

Choice of semantic is implementation-defined

- Can be chosen at compile time, e.g. `-fcontract-semantic=enforce`
- Can be chosen at load time
- Can be chosen at link time
- Can be chosen at runtime
  - e.g. enable assertions when a debugger is attached

→ P3321 ("Contracts interaction with tooling")

Choice of semantic is implementation-defined

- Can be the same for all contract assertions in a translation unit
- Each contract assertion can have a different semantic
  - check only **`pre`**, not **`post`**
  - check only a random subset
- Each evaluation of the same assertion can have a different semantic
  - check every 10th time
- Evaluations can be repeated
  - check pre twice: caller-side & callee-side
    (use case: link binaries together irrespective of contract mode they're compiled with)
- Evaluations can be elided if the compiler can statically prove the result of the check

Consequence: can't rely on side effects

```
#include <cassert>
#ifndef NDEBUG
  unsigned nIter = 0;
#endif
  while (keepIterating()) {
    assert(++nIter < maxIter);
    // ...
  }
```

Consequence: can't rely on side effects

```
#include <cassert>
#ifndef NDEBUG
  unsigned nIter = 0;
#endif
  while (keepIterating()) {
    assert(++nIter < maxIter);
    // ...
  }
```

```
// can't (yet) have code conditional on
// whether contract checks are enabled
unsigned nIter = 0;

while (keepIterating()) {
 contract_assert(++nIter < maxIter);
  // ...          ^^^^^^^ may break!
}
```

Consequence: can't rely on side effects

```
#include <cassert>
#ifndef NDEBUG
  unsigned nIter = 0;
#endif
  while (keepIterating()) {
    assert(++nIter < maxIter);
    // ...
  }
```

```
// can't (yet) have code conditional on
// whether contract checks are enabled
unsigned nIter = 0;
while (keepIterating()) {
  ++nIter;
  contract_assert(nIter < maxIter);
  // ...
}
```

The Contracts Prime Directive

- The presence or evaluation of a contract assertion in a program should not alter the correctness of that program

The Contracts Prime Directive

- The presence or evaluation of a contract assertion in a program should not alter the correctness of that program
- Statically enforced:
  - Adding a contract assertion can't affect Concepts / overload resolution / noexcept operator / SFINAE / if constexpr ...
  - Adding a contract assertion that is not checked can't cause runtime overhead

The Contracts Prime Directive

- The presence or evaluation of a contract assertion in a program should not alter the correctness of that program
- Statically enforced:
  - Adding a contract assertion can't affect Concepts / overload resolution / noexcept operator / SFINAE / if constexpr ...
  - Adding a contract assertion that is not checked can't cause runtime overhead
- Responsibility of the user:
  - Don't use predicates with side effects that can alter the correctness of the program, the result of another contract assertion, or a subsequent check of the same contract assertion

The Contracts Prime Directive

- The presence or evaluation of a contract assertion in a program should not alter the correctness of that program
- Statically enforced:
  - Adding a contract assertion can't affect Concepts / overload resolution / noexcept operator / SFINAE / if constexpr ...
  - Adding a contract assertion that is not checked can't cause runtime overhead
- Responsibility of the user:
  - Don't use predicates with side effects that can alter the correctness of the program, the result of another contract assertion, or a subsequent check of the same contract assertion

The Contracts Prime Directive

- Benefit: You can't get "Heisenbugs"
  (e.g. bugs appearing/disappearing when you enable/disable a contract check)

New in P2900 Revision 8: virtual function support!

Contracts and indirect calls

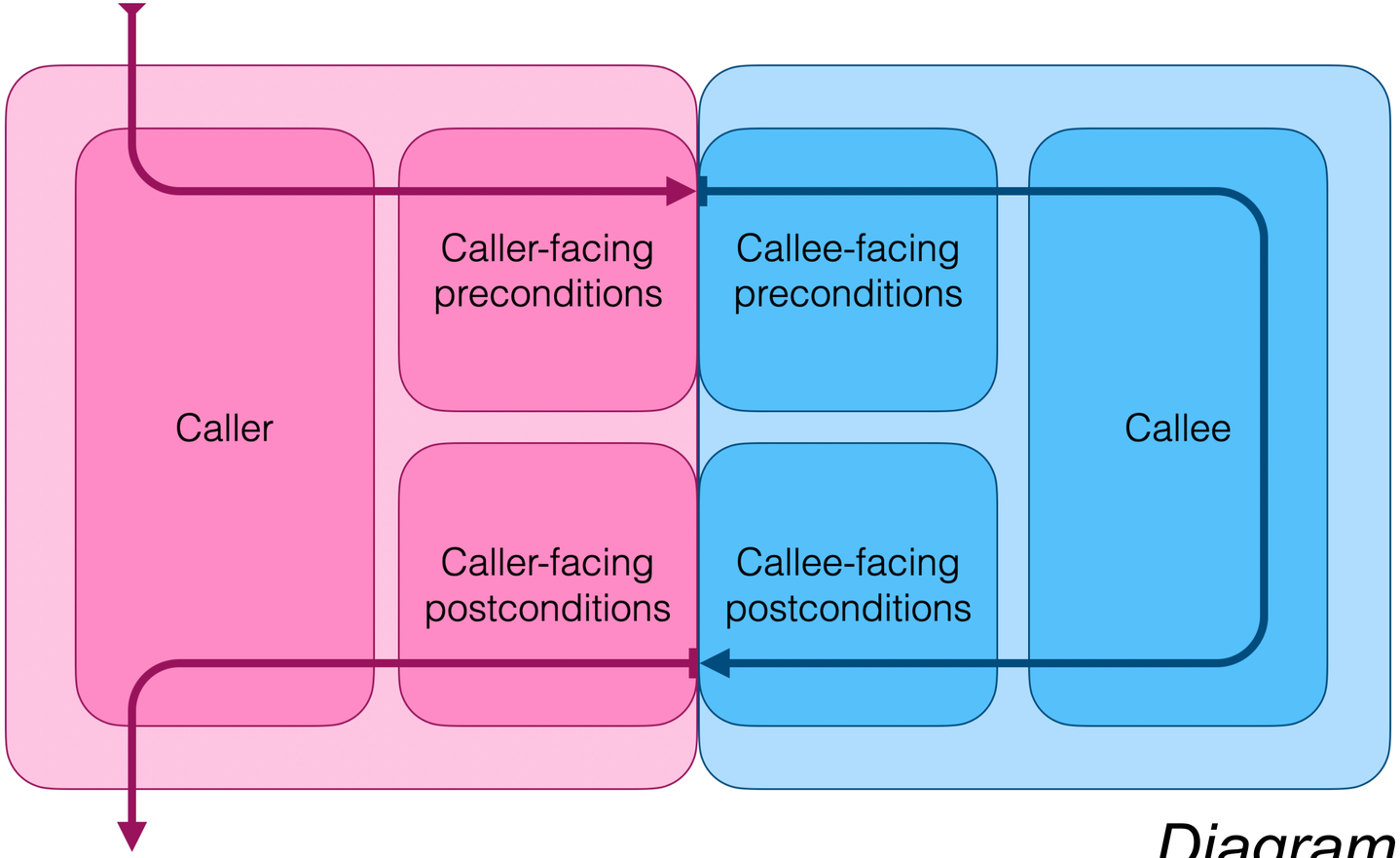


Diagram by Lisa Lippincott

Contracts and virtual function calls

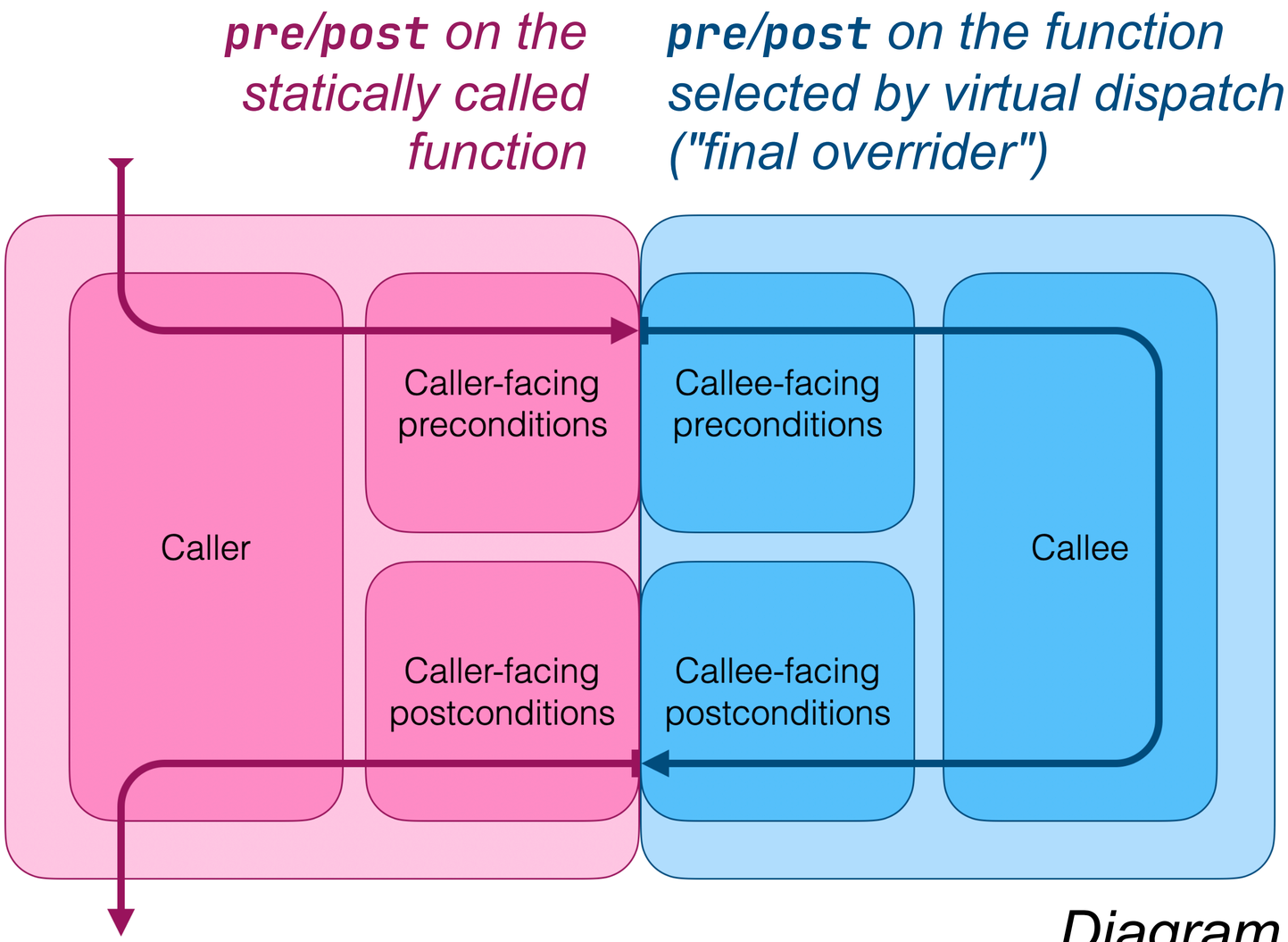


Diagram by Lisa Lippincott

Contracts and virtual function calls

```
struct UnaryFunction {
  virtual Value compute(ArgList args)
    pre (args.size() == 1);
};

struct BinaryFunction {
  virtual Value compute(ArgList args)
    pre (args.size() == 2);
};
```

Diagram by Lisa Lippincott

Contracts and virtual function calls

```
struct UnaryFunction {
  virtual Value compute(ArgList args)
    pre (args.size() == 1);
};

struct BinaryFunction {
  virtual Value compute(ArgList args)
    pre (args.size() == 2);
};

struct VariadicFunction
: UnaryFunction, BinaryFunction {
  Value compute(ArgList args) override
    /* no preconditions */;
};
```

Diagram by Lisa Lippincott

Contracts and virtual function calls

```
struct UnaryFunction {
  virtual Value compute(ArgList args)
    pre (args.size() == 1);
};

struct BinaryFunction {
  virtual Value compute(ArgList args)
    pre (args.size() == 2);
};

struct VariadicFunction
: UnaryFunction, BinaryFunction {
  Value compute(ArgList args) override
    /* no preconditions */;
};
```

```
int main() {
    VariadicFunction varFunc;
    test(varFunc);
}

void test(VariadicFunction& varFunc) {
    varFunc.compute({1});          // OK
    varFunc.compute({2, 3});       // OK
    varFunc.compute({4, 5, 6});    // OK
}
```

Diagram by Lisa Lippincott

Contracts and virtual function calls

```
struct UnaryFunction {
  virtual Value compute(ArgList args)
    pre (args.size() == 1);
};

struct BinaryFunction {
  virtual Value compute(ArgList args)
    pre (args.size() == 2);
};

struct VariadicFunction
: UnaryFunction, BinaryFunction {
  Value compute(ArgList args) override
    /* no preconditions */;
};
```

```
int main() {
    VariadicFunction varFunc;
    test(varFunc);
}

void test(UnaryFunction& unaryFunc) {
    unaryFunc.compute({1});       // OK
    unaryFunc.compute({2, 3});    // violation
    unaryFunc.compute({4, 5, 6}); // violation
}
```

Diagram by Lisa Lippincott

Contracts and virtual function calls

```
struct UnaryFunction {
  virtual Value compute(ArgList args)
    pre (args.size() == 1);
};

struct BinaryFunction {
  virtual Value compute(ArgList args)
    pre (args.size() == 2);
};

struct VariadicFunction
: UnaryFunction, BinaryFunction {
  Value compute(ArgList args) override
    /* no preconditions */;
};
```

```
int main() {
    VariadicFunction varFunc;
    test(varFunc);
}

void test(BinaryFunction& binFunc) {
    binFunc.compute({1});          // violation
    binFunc.compute({2, 3});       // OK
    binFunc.compute({4, 5, 6});    // violation
}
```

Diagram by Lisa Lippincott

Contracts and virtual function calls

catches bugs due to wrong use of interface

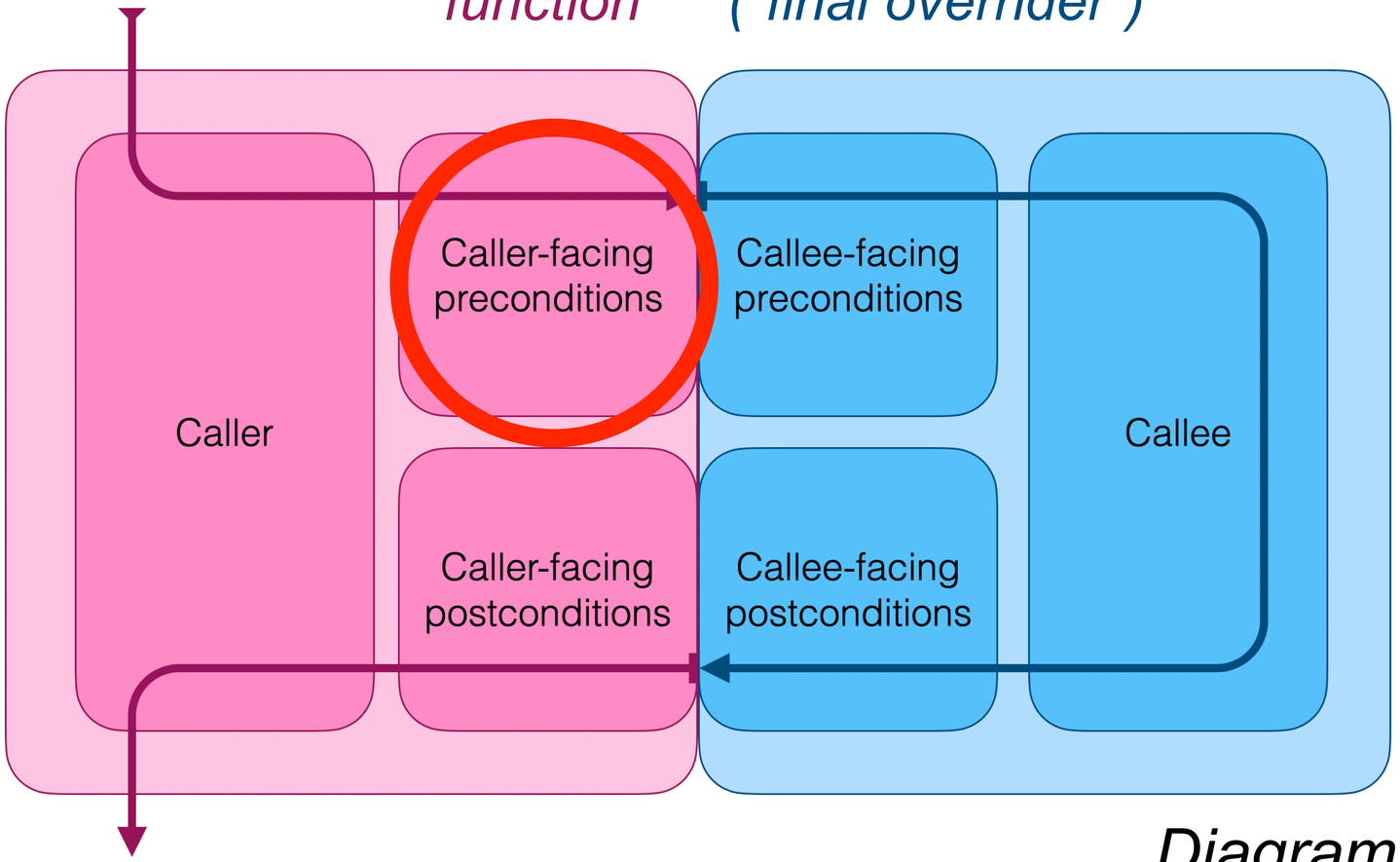
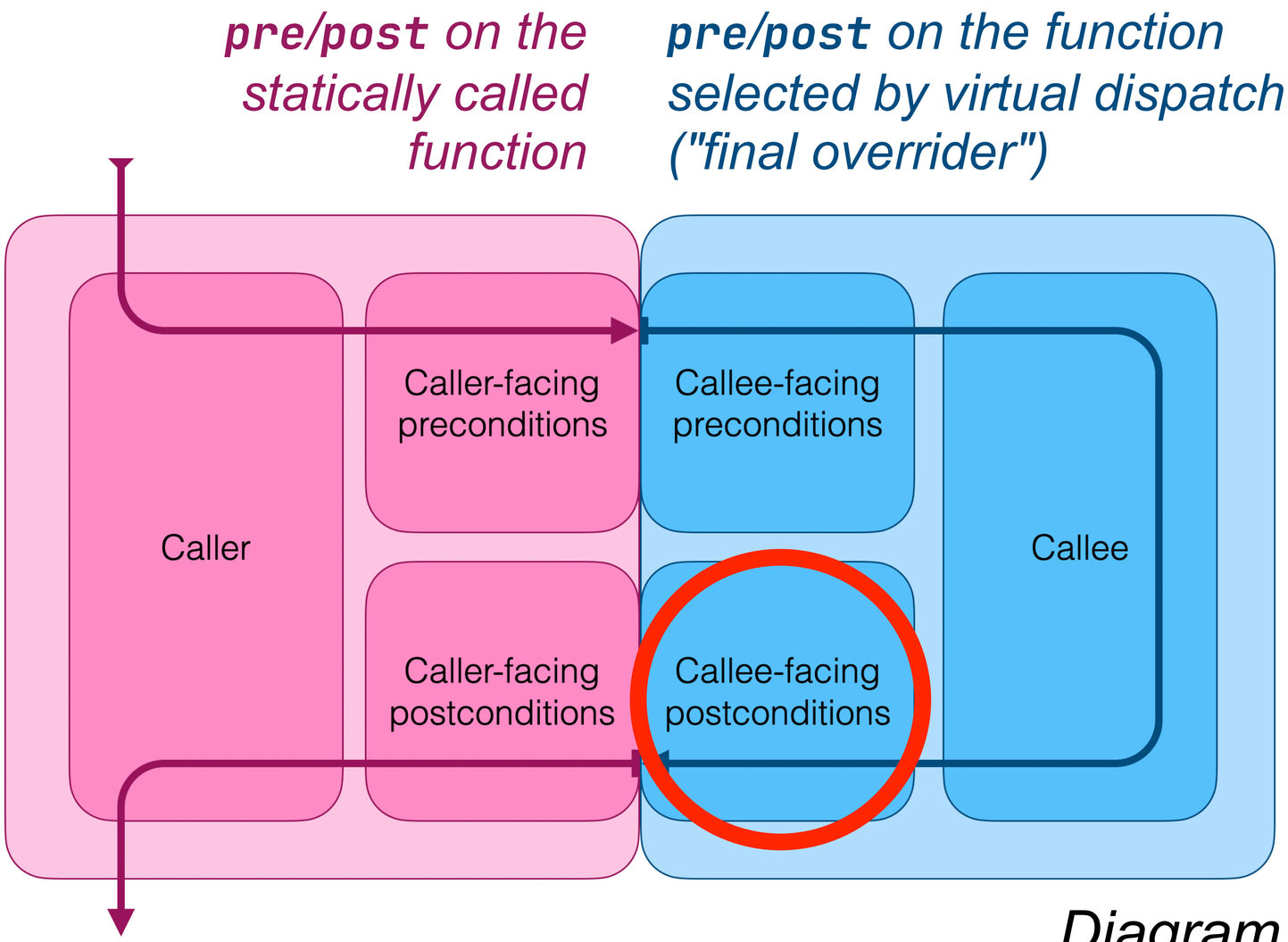


Diagram by Lisa Lippincott

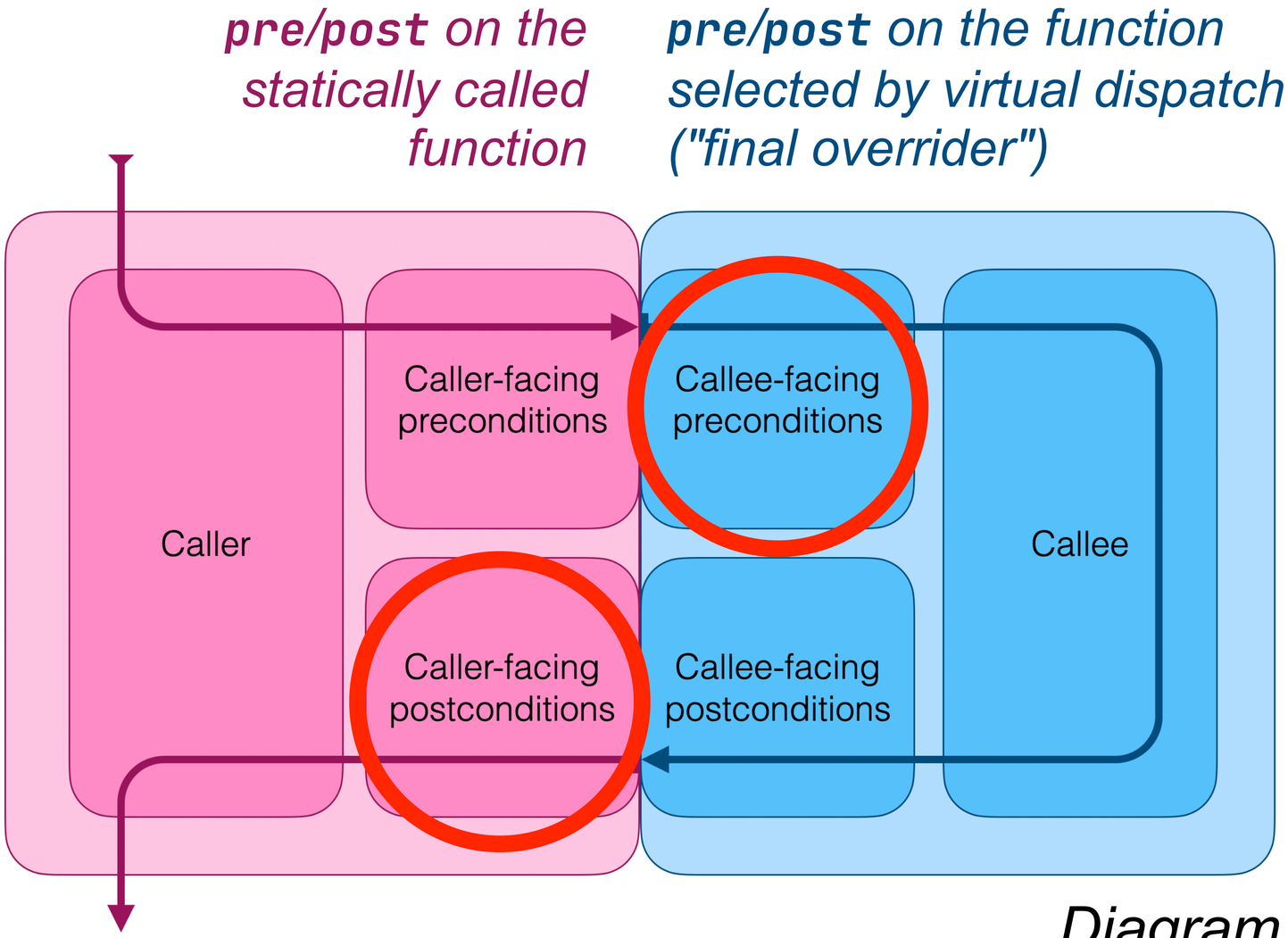
Contracts and virtual function calls



catches bugs in the imple-mentation

Diagram by Lisa Lippincott

Contracts and virtual function calls



catch bugs due to violations of Base/Derived substitutability

Diagram by Lisa Lippincott

Contracts and virtual function calls

→ P3097

→ Lisa Lippincott: "Perspectives on Contracts" (CppCon 2024)

Remaining open questions for P2900

Remaining open questions for P2900

- make `pre` / `post` work on coroutines?

Remaining open questions for P2900

- make **`pre`** / **`post`** work on coroutines?
- make **`pre`** / **`post`** work on function pointers?

Contracts and function pointers

```
int f(int i);

int (*fptr)(int i) pre (i >= 0) = f;
```

Contracts and indirect calls

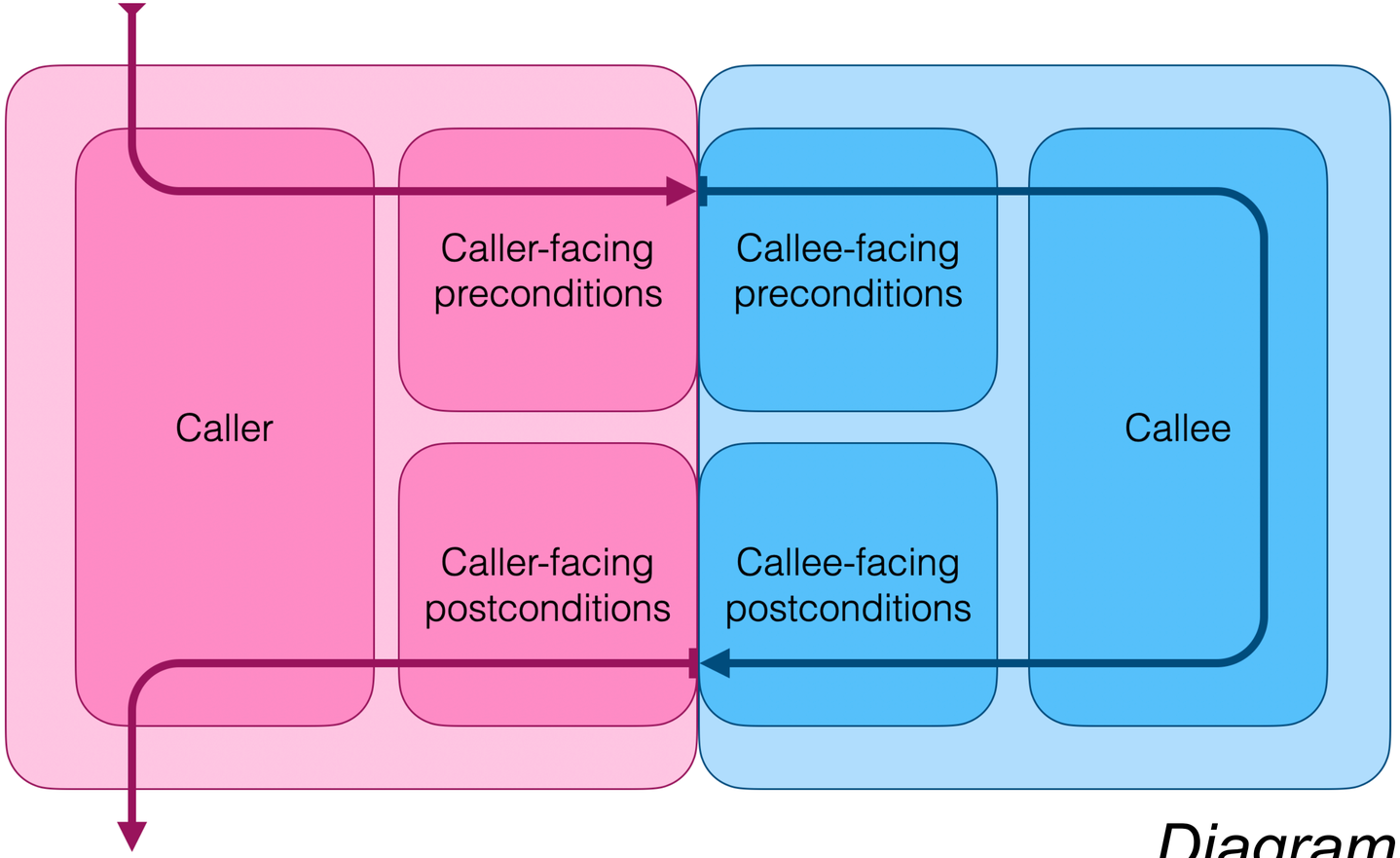


Diagram by Lisa Lippincott

Contracts and indirect calls

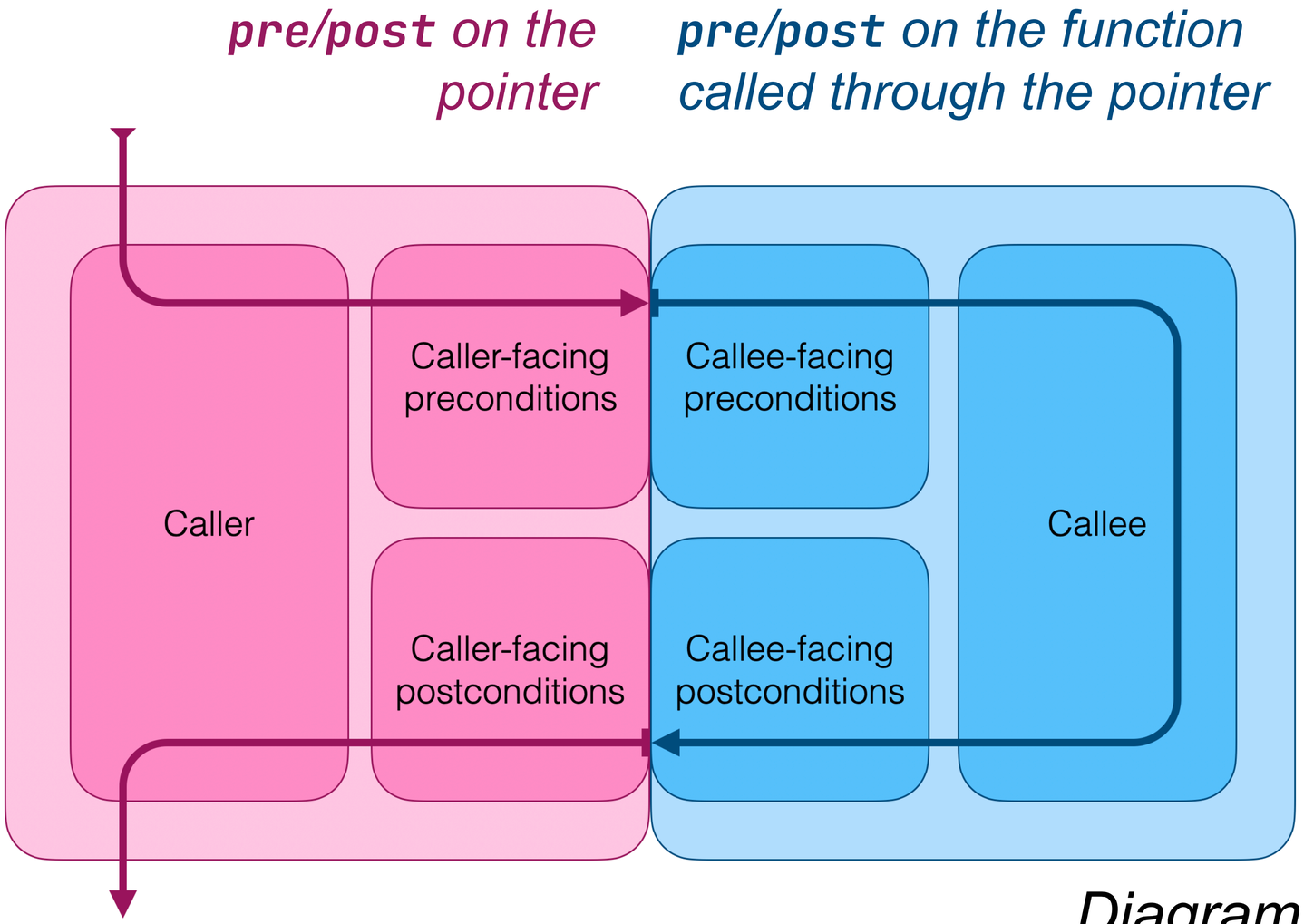


Diagram by Lisa Lippincott

Contracts and function pointers

```
int f(int i);

int (*fptr)(int i) pre (i >= 0) = f;
```

Contracts and function pointers

```
int f(int i);

int (*fptr)(int i) pre (i >= 0) = f;
```

Where does this information live?

In the type? → Code bloat (templates!), code breakage, name mangling (mangle arbitrary C++ expressions?), interaction with overloading?

In the pointer value? → Runtime overhead in space and time, does not work with typedefs, conceptually unsound (contracts are a compile-time property)

In the AST ("property of the declaration")? → Does not work with non-trivial call expressions, typedefs, or templates, does not travel across TU boundaries

Contracts and function pointers

```
int f(int i);

int (*fptr)(int i) pre (i >= 0) = f;
```

Where does this information live?

In the type? → Code bloat (templates!), code breakage, name mangling (mangle arbitrary C++ expressions?), interaction with overloading?

In the pointer value? → Runtime overhead in space and time, does not work with typedefs, conceptually unsound (contracts are a compile-time property)

In the AST ("property of the declaration")? → Does not work with non-trivial call expressions, typedefs, or templates, does not travel across TU boundaries

→ P3271

Remaining open questions for P2900

- make **`pre`** / **`post`** work on coroutines?
- make **`pre`** / **`post`** work on function pointers?
- Keep constification?

Constification

```
int f(int i)
  pre (++i);  // Error: parameters & local variables in contract
              // predicate are implicitly `const`
```

Constification

```
class Widget() {
  // ...
public:
  bool isValid();  // forgot `const`
};
```

Constification

```
class Widget() {
  // ...
public:
  bool isValid();  // forgot `const`
};

Widget getWidget()
  post (w: w.isValid());  // Error: `w` is implicitly `const`
```

Constification

Pros:

- Prevents bugs in new code
- Finds bugs in old code when migrating from macros to **`pre`**/**`post`**/**`contract_assert`**
  - with not much code breakage (case studies: P3268, P3336)

Constification

Pros:

- Prevents bugs in new code
- Finds bugs in old code when migrating from macros to **`pre`**/**`post`**/**`contract_assert`**
  - with not much code breakage (case studies: P3268, P3336)

Cons:

- Can't call functions that are OK to use in **`pre`**/**`post`**/**`contract_assert`** but not marked **`const`**
- Workaround (**`const_cast`**) is ugly and can't always be used
- Expressions inside and outside of **`pre`**/**`post`**/**`contract_assert`** can have different meaning: different overloads selected (**`const`**/non-**`const`**)

Remaining open questions for P2900

- make **`pre`** / **`post`** work on coroutines?
- make **`pre`** / **`post`** work on function pointers?
- Keep constification?
- Undefined behaviour in contract predicates?

Undefined behaviour

```
int f(int a) {
  return a + 100;
}

int g(int a)
  pre (f(a) > a);
```

(example from P2680 - Gabriel Dos Reis)

Undefined behaviour

```
int f(int a) {
  return a + 100;  // compiler can assume this never overflows
}

int g(int a)
  pre (f(a) > a);
```

(example from P2680 - Gabriel Dos Reis)

Undefined behaviour

```
int f(int a) {
  return a + 100;  // compiler can assume this never overflows
}

int g(int a)
  pre (f(a) > a);  // compiler can replace this with `pre (true)`
```

(example from P2680 - Gabriel Dos Reis)

Remaining open questions for P2900

- make `pre` / `post` work on coroutines?
- make `pre` / `post` work on function pointers?
- Keep constification?
- Undefined behaviour in contract predicates?
- Implementation experience?

Remaining open questions for P2900

- make **`pre`** / **`post`** work on coroutines?
- make **`pre`** / **`post`** work on function pointers?
- Keep constification?
- Undefined behaviour in contract predicates?
- Implementation experience?
  - GCC and Clang implementations of P2900 in active development

Possible post-MVP extensions

- Ability to refer to "old" values (at the time of call) inside a postcondition predicate (→ P2461, P3098)

```
void push_back(T& item)
  post [oldSize = size()] (size() == oldSize + 1);
```

- Optimise based on assumption that predicate evaluates to true; otherwise, the behaviour is undefined (a*ssume* semantic)
- Constrain the possible evaluation semantics in code ("labels", "contract levels", "explicit semantics" → P2755)
- **`pre`** / **`post`** on function pointers ("function usage types" → P3271)
- Contract assertions that cannot be expressed by boolean predicates ("procedural interfaces" → P0465)
- Invariants (class invariants, loop invariants...)

What are P2900 Contracts for?

Are P2900 Contracts the solution to safety & security in C++?

P2900 Contracts vs. safety & security

- Contract assertions can significantly improve correctness & safety of code

P2900 Contracts vs. safety & security

- Contract assertions can significantly improve correctness & safety of code,
  - but you have to actually add them to your code!

```
T& operator[] (size_t index) const
  pre (index < size());
```

Checking contracts with contact assertions

- Sometimes straightforward

```
T& operator[] (size_t index) const
  pre (index < size());
```

Checking contracts with contact assertions

- Sometimes straightforward

```
T& operator[] (size_t index) const
  pre (index < size());
```

- Sometimes expensive, or even violates guarantees

```
void binary_search(Iter begin, Iter end)  // O(log N)
  pre (is_sorted(begin, end));            // O(N)
```

Checking contracts with contact assertions

- Sometimes straightforward

```
T& operator[] (size_t index) const
  pre (index < size());
```

- Sometimes expensive, or even violates guarantees

```
void binary_search(Iter begin, Iter end)  // O(log N)
  pre (is_sorted(begin, end));            // O(N)
```

- Sometimes impractical or impossible
  ("`ptr` points to an existing object", "this function will return a value")

P2900 Contracts vs. safety & security

- Contract assertions can significantly improve correctness & safety of code,
  - but you have to actually add them to your code!
- Contract assertions can detect if a contract was violated,
  - but not prove that no contract was violated!
    (only a subset of the plain-language contract can be expressed in code)

P2900 Contracts vs. safety & security

- Contract assertions can significantly improve correctness & safety of code,
  - but you have to actually add them to your code!
- Contract assertions can detect if a contract was violated,
  - but not prove that no contract was violated!
    (only a subset of the plain-language contract can be expressed in code)
- Contract assertions check for correctness during evaluation of the program
  - includes constant evaluation

assertions and constexpr

```
constexpr size_t f(size_t i) pre (i != 0);

std::array<int, f(0)> a;    // Error: violating precondition of `f`
                    ~~~~
```

P2900 Contracts vs. safety & security

- Contract assertions can significantly improve correctness & safety of code,
  - but you have to actually add them to your code!
- Contract assertions can detect if a contract was violated,
  - but not prove that no contract was violated!
    (only a subset of the plain-language contract can be expressed in code)
- Contract assertions check for correctness during evaluation of the program
  - includes constant evaluation
- Contract assertions do not change the semantics of the language
  - No new language constraints
  - No new language guarantees (e.g. guaranteed memory safety)

P2900 Contracts vs. safety & security

- Contract assertions can significantly improve correctness & safety of code,
  - but you have to actually add them to your code!
- Contract assertions can detect if a contract was violated,
  - but not prove that no contract was violated!
    (only a subset of the plain-language contract can be expressed in code)
- Contract assertions check for correctness during evaluation of the program
  - includes constant evaluation
- Contract assertions do not change the semantics of the language
  - No new language constraints
  - No new language guarantees (e.g. guaranteed memory safety)
  - They complement features that do (borrow checker, erroneous values...)

Thank you!