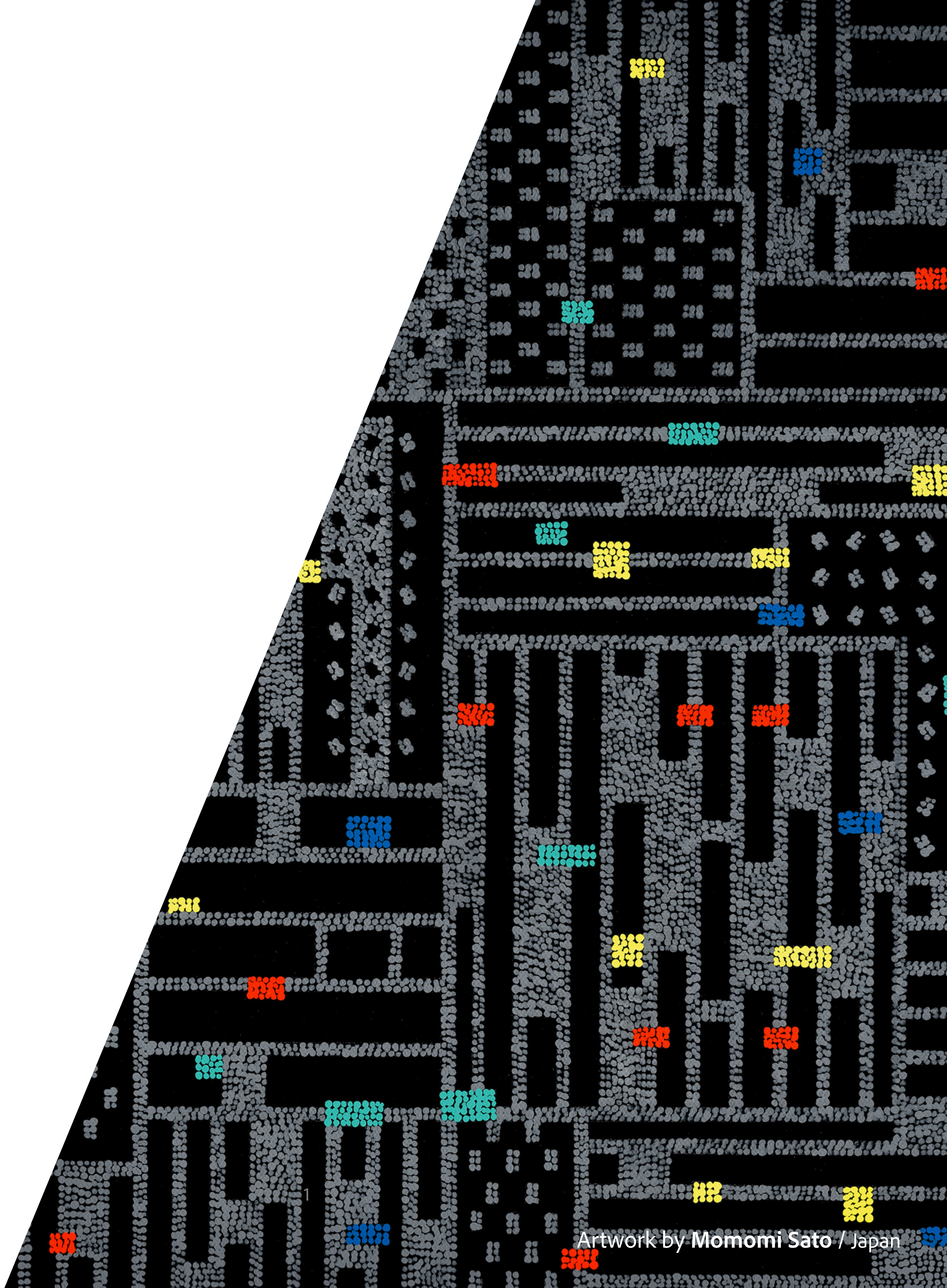
Warning: std::find() is broken!

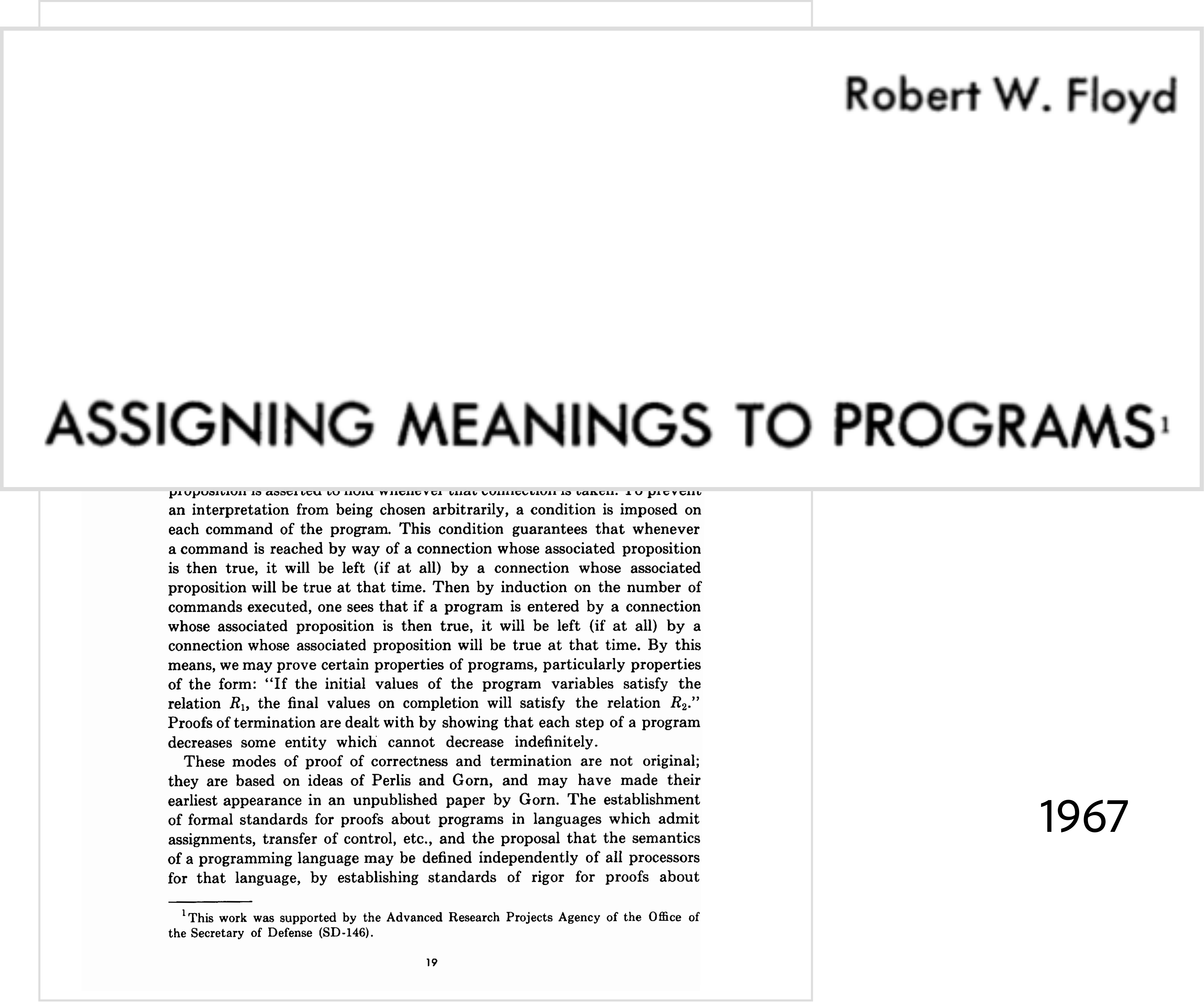
Sean Parent | Sr. Principal Scientist, STLab



Artwork by Momomi Sato / Japan

“Understanding why software fails is important, but the real challenge is understanding why software works.”

– Alexander Stepanov



Robert W. Floyd

ASSIGNING MEANINGS TO PROGRAMS[1]

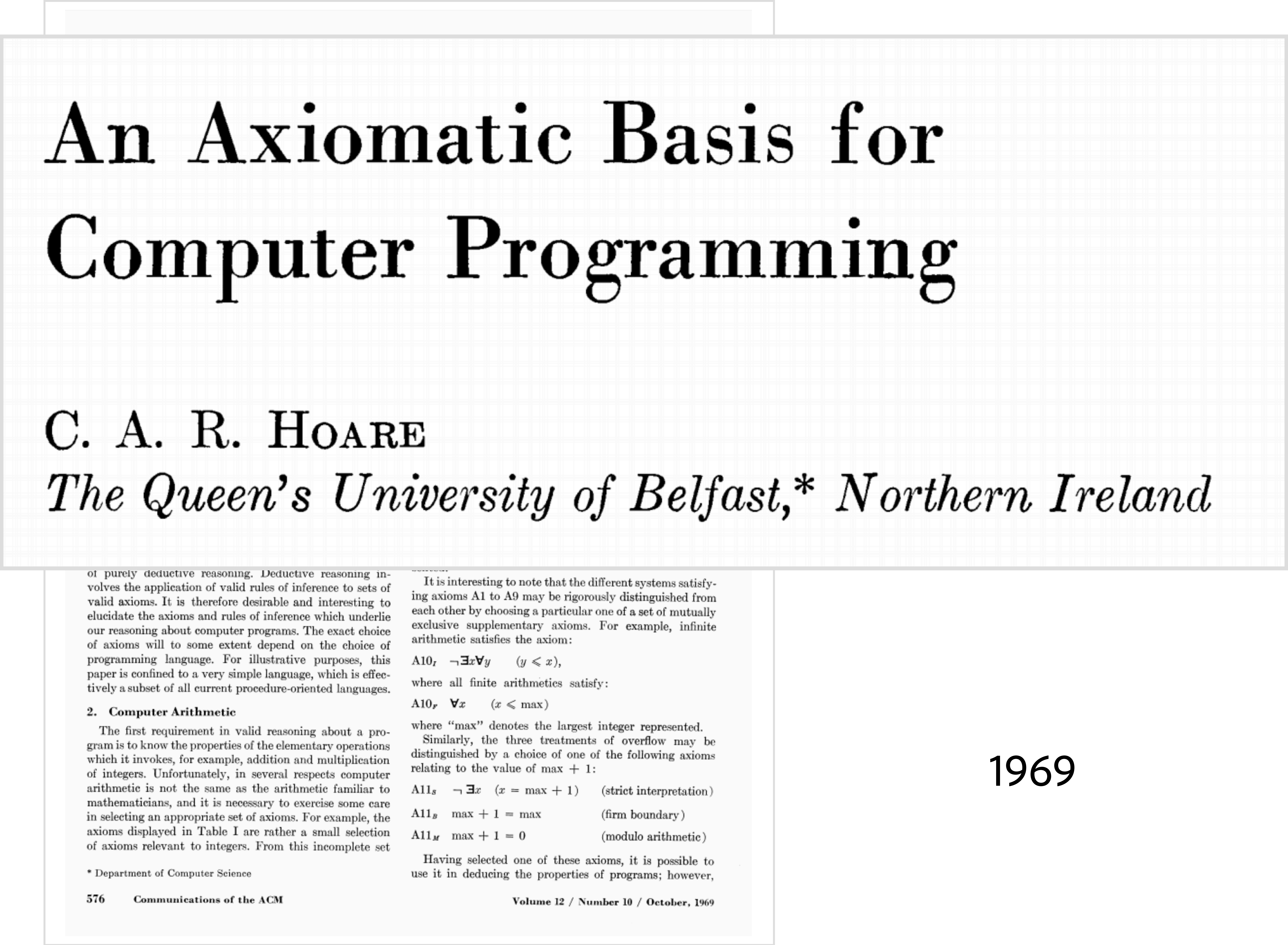
proposition is asserted to hold whenever that connection is taken. To prevent an interpretation from being chosen arbitrarily, a condition is imposed on each command of the program. This condition guarantees that whenever a command is reached by way of a connection whose associated proposition is then true, it will be left (if at all) by a connection whose associated proposition will be true at that time. Then by induction on the number of commands executed, one sees that if a program is entered by a connection whose associated proposition is then true, it will be left (if at all) by a connection whose associated proposition will be true at that time. By this means, we may prove certain properties of programs, particularly properties of the form: "If the initial values of the program variables satisfy the relation R_1, the final values on completion will satisfy the relation R_2." Proofs of termination are dealt with by showing that each step of a program decreases some entity which cannot decrease indefinitely.

These modes of proof of correctness and termination are not original; they are based on ideas of Perlis and Gorn, and may have made their earliest appearance in an unpublished paper by Gorn. The establishment of formal standards for proofs about programs in languages which admit assignments, transfer of control, etc., and the proposal that the semantics of a programming language may be defined independently of all processors for that language, by establishing standards of rigor for proofs about

[1] This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (SD-146).

19

1967



An Axiomatic Basis for Computer Programming

C. A. R. Hoare
The Queen's University of Belfast, Northern Ireland*

of purely deductive reasoning. Deductive reasoning involves the application of valid rules of inference to sets of valid axioms. It is therefore desirable and interesting to elucidate the axioms and rules of inference which underlie our reasoning about computer programs. The exact choice of axioms will to some extent depend on the choice of programming language. For illustrative purposes, this paper is confined to a very simple language, which is effectively a subset of all current procedure-oriented languages.

2. Computer Arithmetic

The first requirement in valid reasoning about a program is to know the properties of the elementary operations which it invokes, for example, addition and multiplication of integers. Unfortunately, in several respects computer arithmetic is not the same as the arithmetic familiar to mathematicians, and it is necessary to exercise some care in selecting an appropriate set of axioms. For example, the axioms displayed in Table I are rather a small selection of axioms relevant to integers. From this incomplete set

It is interesting to note that the different systems satisfying axioms A1 to A9 may be rigorously distinguished from each other by choosing a particular one of a set of mutually exclusive supplementary axioms. For example, infinite arithmetic satisfies the axiom:

$$\text{A10}_I \quad \neg\exists x \forall y \qquad (y \leqslant x),$$

where all finite arithmetics satisfy:

$$\text{A10}_F \quad \forall x \qquad (x \leqslant \max)$$

where "max" denotes the largest integer represented.

Similarly, the three treatments of overflow may be distinguished by a choice of one of the following axioms relating to the value of max + 1:

$$\text{A11}_S \quad \neg\,\exists x \quad (x = \max + 1) \qquad \text{(strict interpretation)}$$

$$\text{A11}_B \quad \max + 1 = \max \qquad \text{(firm boundary)}$$

$$\text{A11}_M \quad \max + 1 = 0 \qquad \text{(modulo arithmetic)}$$

Having selected one of these axioms, it is possible to use it in deducing the properties of programs; however,

* Department of Computer Science

576 Communications of the ACM

Volume 12 / Number 10 / October, 1969

1969

Reasoning About Code: Hoare Logic

Hoare logic, also known as *Floyd-Hoare logic*, describes computation statements as a *Hoare triple*

$$P\{Q\}R\,.$$

Where P is a precondition, Q is an operation, and R is the postcondition.

Statements are combined with rules for assignment, consequence, composition, and iteration.

Given a sequence of statements and assuming an initial precondition, if we can show that the subsequent postconditions guarantee subsequent preconditions are satisfied, then the program is correct.

Math Notation Glossary

for all	$\forall$	(universal quantifier)
there exists	$\exists$	(existential quantifier)
in	$\in$	
such that	$\ni$	
not	$\neg$	
implies	$\Longrightarrow$	
iff	$\Longleftrightarrow$	(if and only if)
logical and	$\wedge$	
logical or	$\vee$	

Properties of Addition for Integers ($\mathbb{Z}$)

$\forall a, b, c \in \mathbb{Z}$	$(a + b) + c = a + (b + c)$	(associative)
$\forall a, b \in \mathbb{Z}$	$a + b = b + a$	(cummutative)
$\exists 0 \ni \forall a, 0 \in \mathbb{Z}$	$a + 0 = a$	(additive identity)
$\forall a \in \mathbb{Z}, \exists (\text{-}a) \in \mathbb{Z} \ni$	$a + (\text{-}a) = 0$	(additive inverse)

$\mathbb{Z} \neq$ int

When signed integers overflow or underflow the behavior is undefined.

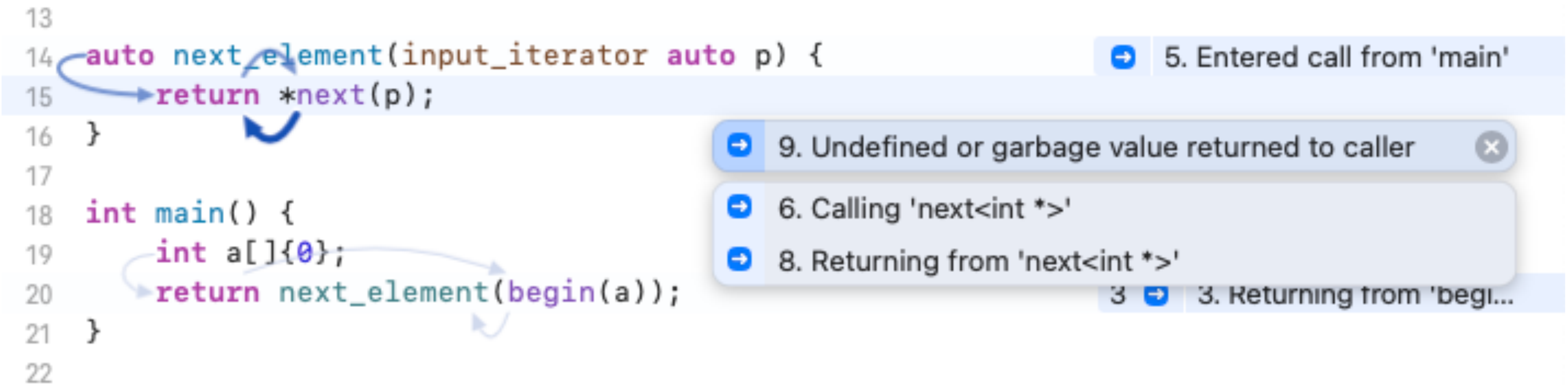
In Hoare logic this could be expressed as an additional axiom:

$$\neg\exists(x \in \text{int}) \ni (x > max_{int})$$

Leading to a Hoare-triple:

$$(a + b \leq max_{int})\{\text{int n = a + b; }\}(n \leq max_{int})$$

“Even the characterization of integer arithmetic is far from complete.”
– C.A.R. Hoare, An Axiomatic Basis for Computer Programming



```
auto next_element(input_iterator auto p) {
    return *next(p);
}

int main() {
    int a[]{0};
    return next_element(begin(a));
}
```

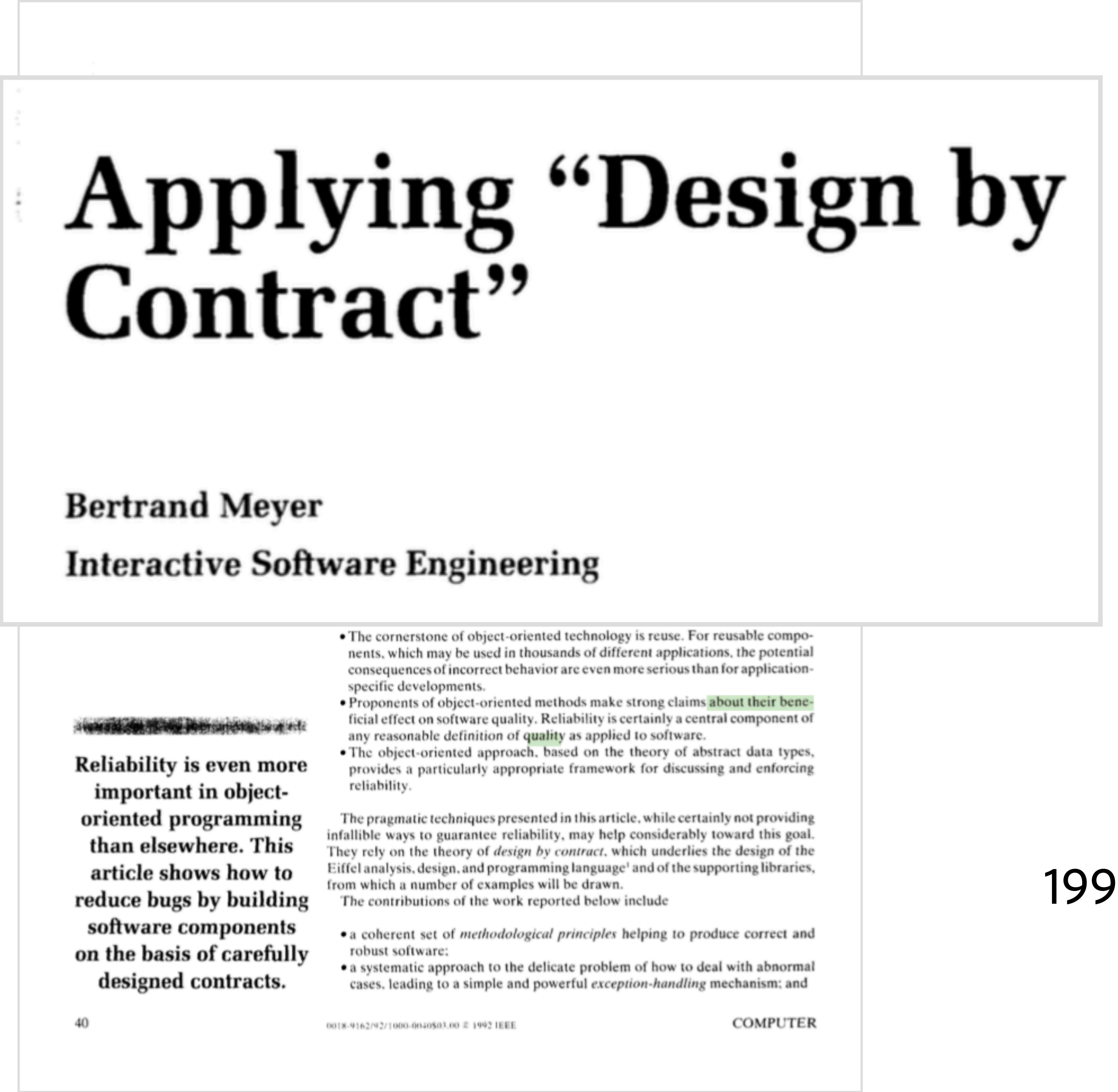
5. Entered call from 'main'

9. Undefined or garbage value returned to caller

6. Calling 'next<int *>'

8. Returning from 'next<int *>'

3 3. Returning from 'begi...



Applying “Design by Contract”

Bertrand Meyer

Interactive Software Engineering

Reliability is even more important in object-oriented programming than elsewhere. This article shows how to reduce bugs by building software components on the basis of carefully designed contracts.

- The cornerstone of object-oriented technology is reuse. For reusable components, which may be used in thousands of different applications, the potential consequences of incorrect behavior are even more serious than for application-specific developments.
- Proponents of object-oriented methods make strong claims about their beneficial effect on software quality. Reliability is certainly a central component of any reasonable definition of quality as applied to software.
- The object-oriented approach, based on the theory of abstract data types, provides a particularly appropriate framework for discussing and enforcing reliability.

The pragmatic techniques presented in this article, while certainly not providing infallible ways to guarantee reliability, may help considerably toward this goal. They rely on the theory of *design by contract*, which underlies the design of the Eiffel analysis, design, and programming language[1] and of the supporting libraries, from which a number of examples will be drawn.

The contributions of the work reported below include

- a coherent set of *methodological principles* helping to produce correct and robust software;
- a systematic approach to the delicate problem of how to deal with abnormal cases, leading to a simple and powerful *exception-handling* mechanism; and

0018-9162/92/1000-0040$03.00 © 1992 IEEE

1992 (original 1986)

Design by Contract

Preconditions and postconditions are asserted in code, in the interface

```
set_minute (m: INTEGER)
        -- Set the minute from `m'.
    require
        valid_argument_for_minute: 0 <= m and m <= 59
    ensure
        minute_set: minute = m
    end
```

Design by Contract

Class invariants define postconditions for all (public) operations on a class

```
invariant
   minute_valid: 0 <= minute and minute <= 59
```

By extension, class invariants define a *guarantee* for any operation taking an instance of the class as an argument

Implementation Limitations of Contracts

Assertions must be expressible in code

Complexity of runtime checked assertions is limited

- A linear time assertion on a constant operation can transform code from $O(n)$ to $O(n \bullet m)$

Cannot validate universal quantifiers, ∀, or existential quantifiers, ∃ without a logical verification system

Key Contributions of Design by Contract

Simplifies formal methods by inverting the process to top-down

- Given a precondition, it is simpler to prove a function satisfies a postcondition than to derive a preconditions and postconditions from a composition of operations

The ideas of design by contract are not limited to implementation constraints

- Assertions that cannot be expressed or validated directly in code can be expressed in documentation

Makes formal methods practical for every programmer

“With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviors of your system will be depended on by somebody.”

– Hyrum Wright

Remove the first odd number (attempt 1):

```
vector a{0, 1, 2, 3, 4, 5};

// Remove the first odd number
auto p = remove_if(begin(a), end(a), [odd_count{0}](int x) mutable {
    return (x & 1) && (++odd_count == 1);
});

a.erase(p, end(a));
display(a);
```

```
 { 0, 2, 4, 5 }
{ 0, 2, 3, 4, 5 }
```

Possible Implementation of std::remove_if()

```
template <class F, class P>
auto remove_if(F f, F l, P pred) {
    f = find_if(f, l, pred); // <-- pred is passed by value

    if (f == l) return f;

    for (auto p = next(f); p != l; ++p) {
        if (!pred(*p)) *f++ = move(*p);
    }
    return f;
}
```

Remove the first odd number (attempt 2):

```cpp
vector a{0, 1, 2, 3, 4, 5};

// Remove the first odd number
int odd_count{0};
auto p = remove_if(begin(a), end(a), [&odd_count](int x) {
    return (x & 1) && (++odd_count == 1);
});

a.erase(p, end(a));
display(a);
```

```
{ 0, 2, 3, 4, 5 }
```

Standard Requirement for Unary Predicate

“Given a glvalue `u` of type (possibly `const`) T that designates the same object as `*first`, `pred(u)` shall be a valid expression that is equal to `pred(*first)`.”

`pred()` is a required to be a *regular* function.

But Hyrum’s Law…

Safety & Correctness

An operation is safe if it cannot lead to undefined behavior

- directly or indirectly
- even if the operation preconditions are violated

an unsafe operation may lead to undefined behavior if preconditions ever are violated

- either directly or during subsequent operations, safe or not

Code that violates preconditions is incorrect

Safety is about the possible consequences of having a bug

Requirements For Correctness

A correctly implemented operation guarantees that:

- If preconditions are satisfied
  - The operation will either succeed, result matches post conditions
  - Or report failure, return an error, throw an exception, set errno
    - Any objects being mutated by the operation must be left in a “known or determinable state.”
      - A weaker requirement than valid.

Requirements For Correctness

If a precondition is not satisfied

- If the operation is safe
  - The result is unspecified which could include:
    - failure (return an error, throw an exception)
    - trapping (calling terminate)
    - leaving any object being mutated by the operation in an unspecified, possibly invalid, state

Requirements For Correctness

If a precondition is not satisfied

- If the operation is unsafe
  - The behavior is undefined
    - If the operation returns, any subsequent operation is also undefined
    - Undefined behavior may including writing to arbitrary memory, executing arbitrary functions, damaging the hardware, launching the missile, crashing the car… anything
    - Compilers are free to assume undefined behavior does not happen

Undefined Behavior

```
#include <iostream>

void function(const int& x) {
    if (&x == nullptr) std::cout << “null-reference\n";
    else std::cout << “valid\n";
}

int main() {
    int* p = nullptr;
    function(*p);
}
```

```
valid
```

Weakening Preconditions

An implementation may do something specifiable and safe when a precondition is violated

- It is tempting to weaken preconditions and specify those cases
  - Because Hyrum…

But should we?

“It’s complicated.”

– Kate Gregory

Signed vs Unsigned Integral Types

Signed integral types in C++ have more preconditions than unsigned types

- It is undefined behavior to overflow or underflow a signed type
- unsigned types are mod(2^b)

Unsigned types still have preconditions

```
{
    unsigned a;
    unsigned b = a; // undefined behavior
}
```

A precondition of reading a variable is that it has been initialized

Bresenham Line Algorithm

```
template <class F>
void bresenham_line(int dx, int dy, F out) {
    assert((0 <= dy) && (dy <= dx) && (dx <= (INT_MAX - dy)));

    for (int x = 0, y = 0, a = dy / 2; x != dx; ++x) {
        out(x, y);
        a += dy;
        if (a >= dx) {
            ++y;
            a -= dx;
        }
    }
}
```

Bresenham Line Algorithm

```
bresenham_line(10, 6, [](auto, auto y) {
    cout << string(y, ' ') << "*\n";
});
```

```
*
*
 *
  *
  *
   *
   *
    *
     *
     *
```

Bresenham Line Algorithm

```
template <class F>
void bresenham_line(int dx, int dy, F out) {
    assert((0 <= dy) && (dy <= dx) && (dx <= (INT_MAX - dy)));

    for (int x = 0, y = 0, a = dy / 2; x != dx; ++x) {
        out(x, y);
        a += dy;
        if (a >= dx) {
            ++y;
            a -= dx;
        }
    }
}
```

Bresenham Line Algorithm

```
a += dy;
if (a >= dx) {

  a -= dx;
}
```

$$a + dy \to a \pmod{dx}$$

Faster Bresenham Line Algorithm

Unsigned integer arithmetic in C++ is mod $2^n = \text{mod}\ (max + 1)$

Project the slope to $dx' = max + 1$

$$\frac{dy}{dx} = \frac{dy'}{(max + 1)}$$

$$dy' = \frac{(max + 1)dy}{dx}$$

Faster Bresenham Line Algorithm - exploiting unsigned

```
template <class F>
void fast_bresenham_line(unsigned dx, unsigned dy, F out) {
    assert(dy < dx);

    dy = (UINT_MAX + 1.0) * dy / dx;

    for (unsigned x = 0, y = 0, a = dy / 2; x != dx; ++x) {
        out(x, y);
        a += dy;                    // add ebx, r12d
        y += unsigned(a < dy); // addc r15d, 0
    }
}
```

Fast Bresenham Line Algorithm

```
fast_bresenham_line(10, 6, [](auto, auto y) {
    cout << string(y, ' ') << "*\n";
});
```

```
*
*
 *
  *
  *
   *
   *
    *
     *
     *
```

Unsigned vs Signed Integers

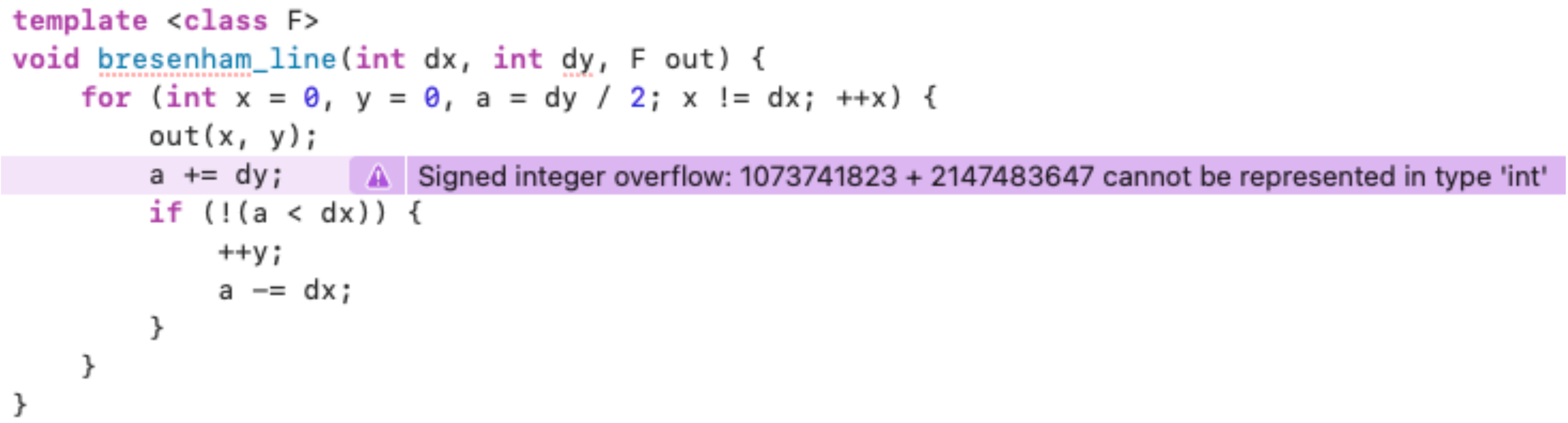
It is easier to detect overflow with modular arithmetic

Modular arithmetic properties can be exploited

Usually math is modeling a subset of $\mathbb{Z}$ or $\mathbb{N}$

- Signed math provides more opportunities for analyzers to detect possible overflow

Undefined Behavior Can Catch Defects



```
template <class F>
void bresenham_line(int dx, int dy, F out) {
    for (int x = 0, y = 0, a = dy / 2; x != dx; ++x) {
        out(x, y);
        a += dy;
        if (!(a < dx)) {
            ++y;
            a -= dx;
        }
    }
}
```

⚠ Signed integer overflow: 1073741823 + 2147483647 cannot be represented in type 'int'

Unsigned vs Signed Integers

- Range of values, and limit behavior: *modulo, trap, saturate, unspecified*
  - *undefined?*
- Could be encoded as separate types (Rust) or separate operations (Swift)

Strong Preconditions

Pros

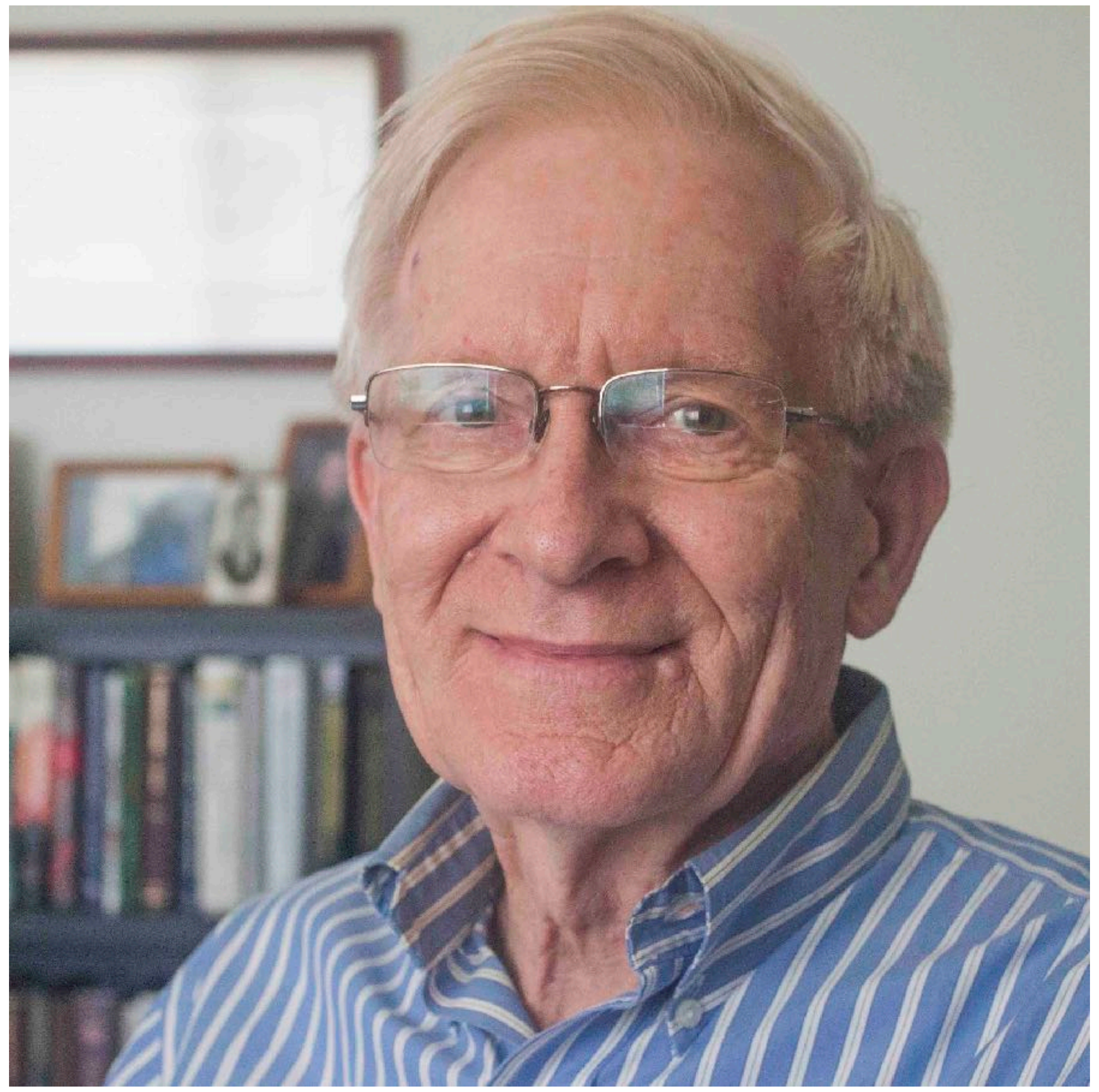
- Provide flexibility of implementation
- Can ascribe meaning and intent to an operation
- Simplify requirements and reasoning about code

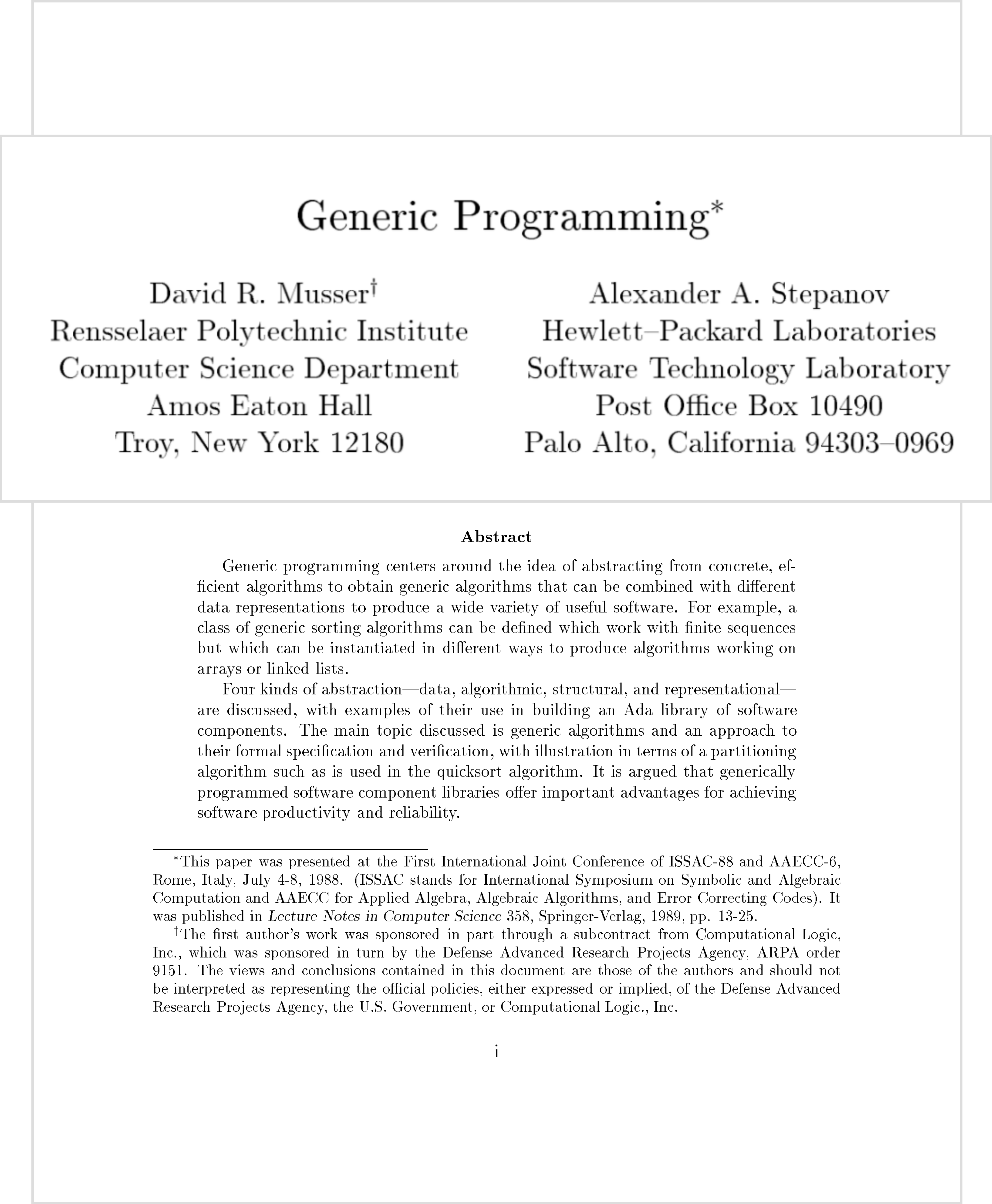
Cons

- Limit clever uses that exploit otherwise defined behavior
- Allow for variance in behavior between implementations
- Open an opportunity for Hyrum’s law

“God created the natural numbers. All else is the work of man.”

– Leopold Kronecker





Generic Programming*

David R. Musser[†]
Rensselaer Polytechnic Institute
Computer Science Department
Amos Eaton Hall
Troy, New York 12180

Alexander A. Stepanov
Hewlett–Packard Laboratories
Software Technology Laboratory
Post Office Box 10490
Palo Alto, California 94303–0969

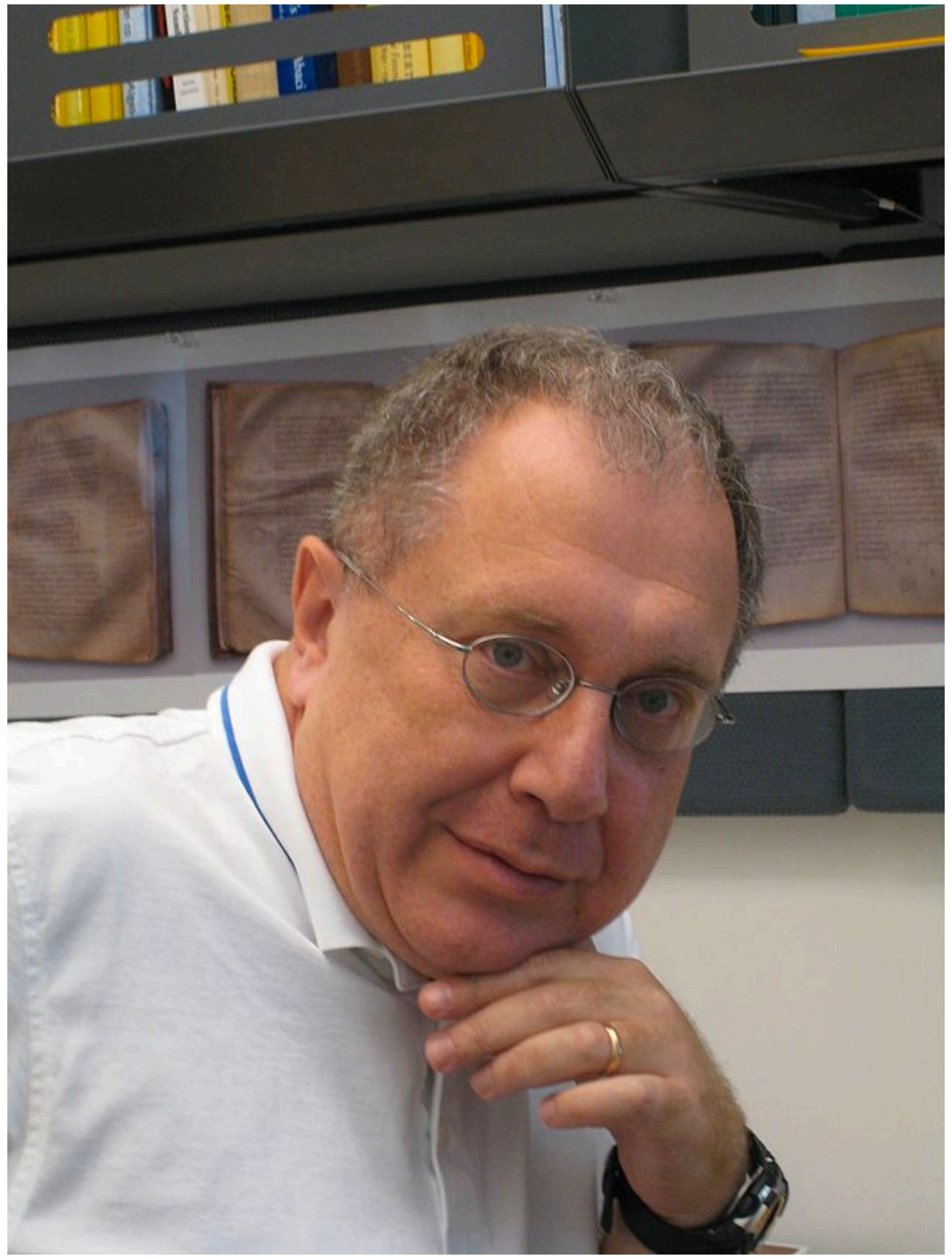
Abstract

Generic programming centers around the idea of abstracting from concrete, efficient algorithms to obtain generic algorithms that can be combined with different data representations to produce a wide variety of useful software. For example, a class of generic sorting algorithms can be defined which work with finite sequences but which can be instantiated in different ways to produce algorithms working on arrays or linked lists.

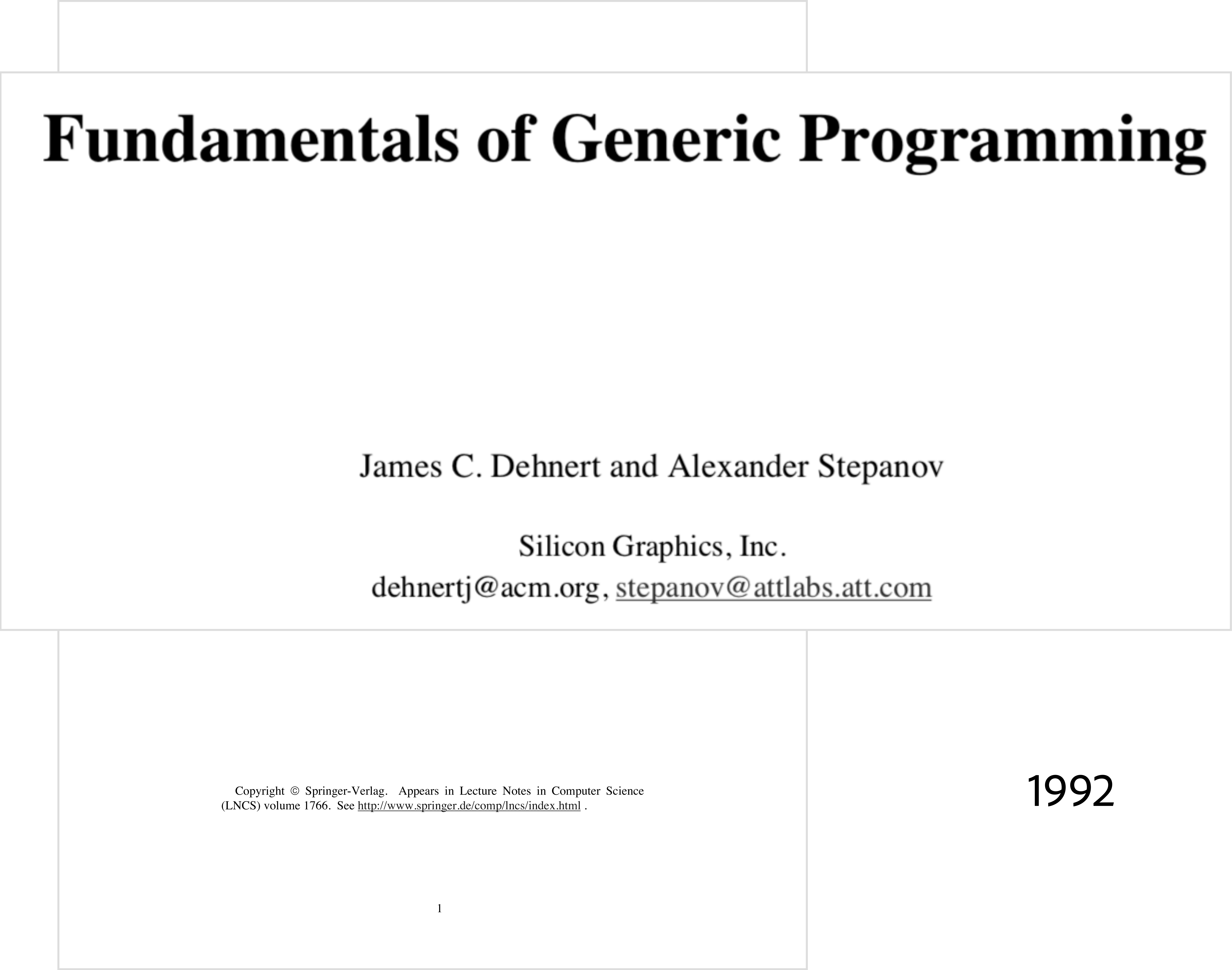
Four kinds of abstraction—data, algorithmic, structural, and representational—are discussed, with examples of their use in building an Ada library of software components. The main topic discussed is generic algorithms and an approach to their formal specification and verification, with illustration in terms of a partitioning algorithm such as is used in the quicksort algorithm. It is argued that generically programmed software component libraries offer important advantages for achieving software productivity and reliability.

*This paper was presented at the First International Joint Conference of ISSAC-88 and AAECC-6, Rome, Italy, July 4-8, 1988. (ISSAC stands for International Symposium on Symbolic and Algebraic Computation and AAECC for Applied Algebra, Algebraic Algorithms, and Error Correcting Codes). It was published in *Lecture Notes in Computer Science* 358, Springer-Verlag, 1989, pp. 13-25.

[†] The first author's work was sponsored in part through a subcontract from Computational Logic, Inc., which was sponsored in turn by the Defense Advanced Research Projects Agency, ARPA order 9151. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the U.S. Government, or Computational Logic., Inc.



1989



Fundamentals of Generic Programming

James C. Dehnert and Alexander Stepanov

Silicon Graphics, Inc.
dehnertj@acm.org, stepanov@attlabs.att.com

1992

"We call the set of axioms satisfied by a data type and a set of operations on it a concept."

– Fundamentals of Generic Programming

Concepts

Concepts are a named set of requirements

- axioms specifying the semantics of operations (semantic requirements)
- operation preconditions and postconditions (contractual requirements)
- operation complexity (complexity requirements)

C++20 Concepts

In C++20, concepts associate a documented set of semantic, contractual, and complexity requirements with a set of named operations (syntactic requirements)

- Similar to how natural language works, we associate meaning with words
- Example: `equality_comparable` requires
  - `operator==` is defined and the result is convertible to bool (syntactic)
  - `operator==` is an equivalence relation (semantic)
  - The arguments to `operator==` are within the domain of the operation (contractual)
  - `operator==` executes in time proportional to the area of the object (complexity)

Concepts

Associate semantics & complexity with syntax

Defines a component that will work for any type satisfying the requirements

Assign meaning to an unbounded set of operations

An argument is required to satisfy a concept

A data type or operation may guarantee it is able to satisfy a concept

Requirements vs Guarantees

A *requirement* applies to the parameters of a (parameterized) type or operation

A *guarantee* applies to an instance of an object, or objects:

- Asserting such an instance satisfies a requirement (or models a concept)

Requirements

`distance(f, l)` requires:

- `f` and `l` satisfy *InputIterators*
  - preincrement, `++i`, postincrement, `(void)i++`, and postincrement and dereference, `*i++`
    - precondition: `i != l`
  - `f` and `l` satisfy *TrivialIterator*
    - `f` and `l` satisfy *Assignable*, *EqualityComparable*, *DefaultConstructible*
      - *EqualityComparable* precondition: arguments are in the domain of ==
- precondition: `[f, l)` is a valid range

Requirements

Naming the set of requirements is a significant simplification

The concept `std::input_iterator` encapsulates a complex set of syntactic and semantic requirements

Only the syntactic requirements are enforced by the compiler but analyzers and sanitizers can validate some of the semantic requirements

Concepts in the standard are requirements on the parameters of the library components

- The standard types are often described as guaranteeing they satisfying some concepts

Concepts

Named requirements, or concepts, are distilled from:

- A set of related components (algorithms, containers, types…)
- A set of common models

They create a simple way to match data types to components and know the result will work correctly

- For a specific component, requirements may be stronger than those required by the implementation

The purpose is not to specify the implementation but to specify the meaning

std::find(first, last, value)

Meaning of Equality

Two objects are equal iff they represent the same entity (i.e., have the same value)

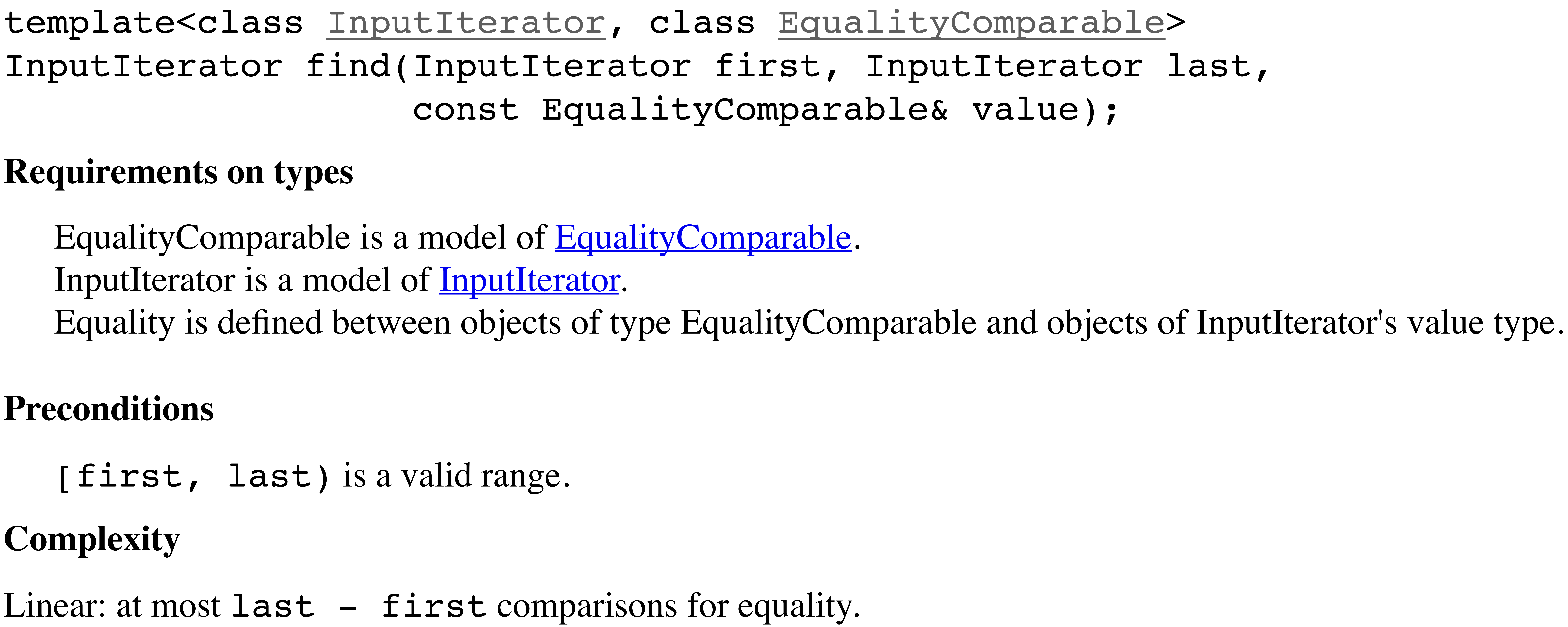
Equality is an equivalence relation

$$
\begin{array}{rrl}
\forall a & a = a & \text{(reflexive)} \\
\forall a, b & a = b \iff b = a & \text{(symmetric)} \\
\forall a, b, c & (a = b \wedge b = c) \implies a = c & \text{(transitive)}
\end{array}
$$

Consistent with other operations on the type

$$
\begin{array}{rrl}
\forall a, b & b \rightarrow a \implies a = b & \text{(equivalence of copies)} \\
\forall a, b & a \nless b \wedge b \nless a \iff a = b & \text{(excluded middle)}
\end{array}
$$

SGI STL std::find() documentation



```
template<class InputIterator, class EqualityComparable>
InputIterator find(InputIterator first, InputIterator last,
                   const EqualityComparable& value);
```

Requirements on types

EqualityComparable is a model of EqualityComparable.
InputIterator is a model of InputIterator.
Equality is defined between objects of type EqualityComparable and objects of InputIterator's value type.

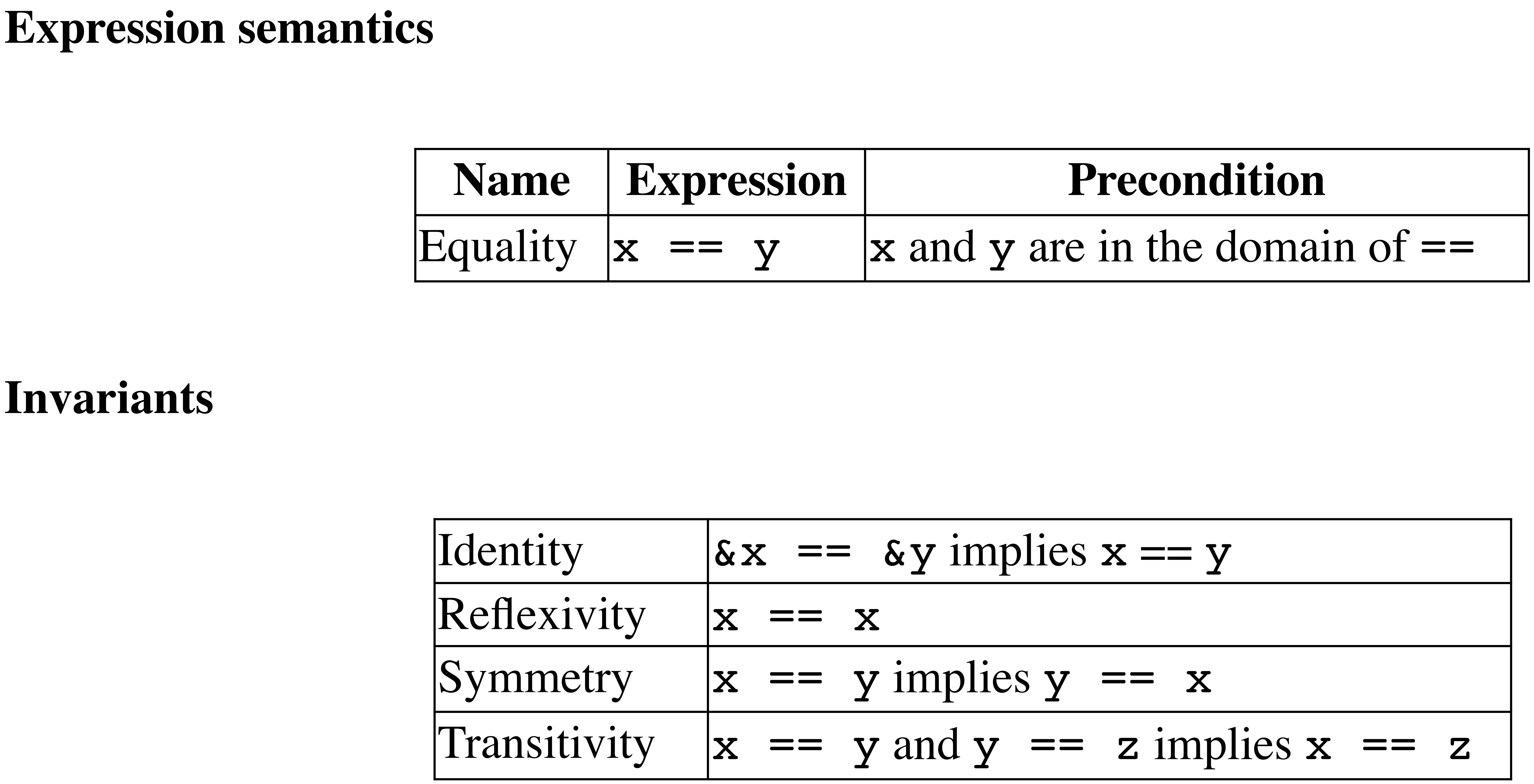
Preconditions

`[first, last)` is a valid range.

Complexity

Linear: at most `last - first` comparisons for equality.

SGI STL EqualityComparable documentation



Expression semantics

Name	Expression	Precondition
Equality	`x == y`	`x` and `y` are in the domain of `==`

Invariants

Identity	`&x == &y` implies `x == y`
Reflexivity	`x == x`
Symmetry	`x == y` implies `y == x`
Transitivity	`x == y` and `y == z` implies `x == z`

C++20 std::find() specification (25.6.5)

```
template<class InputIterator, class T>
  constexpr InputIterator find(InputIterator first, InputIterator last,
                               const T& value);
```

Let *E* be:

`*i == value` for `find`;

Returns: The first iterator `i` in the range `[first, last)` for which *E* is `true`. Returns `last` if no such iterator is found.

C++20 Cpp17EqualityComparable requirements

Table 27: *Cpp17EqualityComparable* requirements [tab:cpp17.equalitycomparable]

Expression	Return type	Requirement
`a == b`	convertible to `bool`	== is an equivalence relation, that is, it has the following properties: — For all a, a == a. — If a == b, then b == a. — If a == b and b == c, then a == c.

NaN refresher - a value that is not *equality comparable*

`nan(“”)` is typically generated by `0.0/0.0`

`nan(“”) == nan(“”)` is `false` (irreflexive)

`nan(“”)` does not satisfy the requirements of EqualityComparable or Cpp17EqualityComparable

Find Without Equality

```
double a[]{ 0.8, 7.0, nan(""), 3.0, 2.4 };

auto p = find(begin(a), end(a), nan(""));

if (p == end(a)) {
    cout << "not-found\n";
} else {
    cout << "found: " << *p << "\n";
}
```

```
not-found
```

Find Without Equality

```cpp
double a[]{ 0.8, 7.0, nan(""), 3.0, 2.4 };

auto p = find(begin(a), end(a), 3.0);

if (p == end(a)) {
    cout << "not-found\n";
} else {
    cout << "found: " << *p << "\n";
}
```

```
found: 3
```

A Subtle Change…

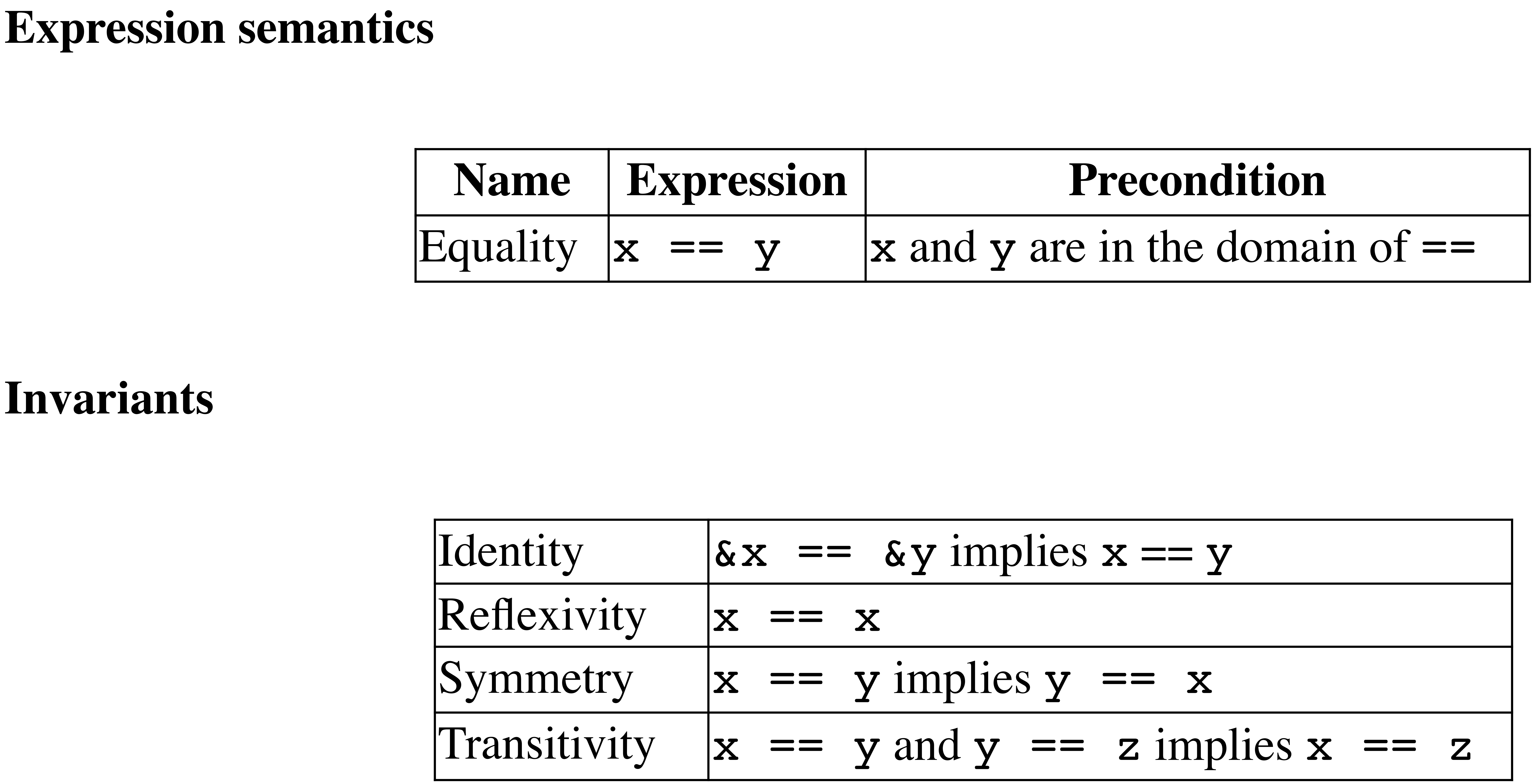
If std::find() required Cpp17EqualityComparable, the following code would be undefined behavior:

```
double a[] { 0.8, 7.0, 42.3, 3.0, 2.4 };

auto p = find(begin(a), end(a), 3.0);
```

The above code is well defined with the SGI definition of EqualityComparable

SGI STL EqualityComparable documentation



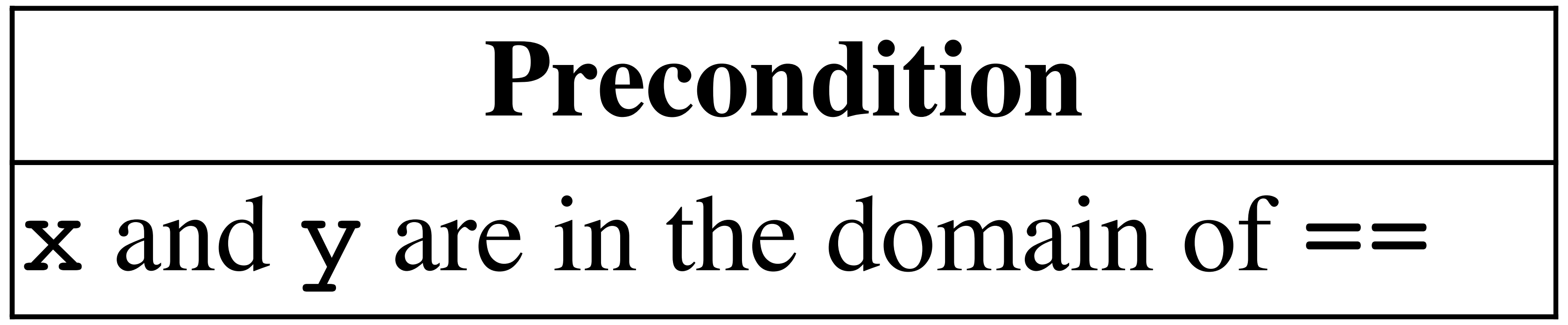
Expression semantics

Name	Expression	Precondition
Equality	`x == y`	`x` and `y` are in the domain of `==`

Invariants

Identity	`&x == &y` implies `x == y`
Reflexivity	`x == x`
Symmetry	`x == y` implies `y == x`
Transitivity	`x == y` and `y == z` implies `x == z`

SGI STL EqualityComparable documentation



Precondition
`x` and `y` are in the domain of `==`

C++20 Cpp17EqualityComparable requirements

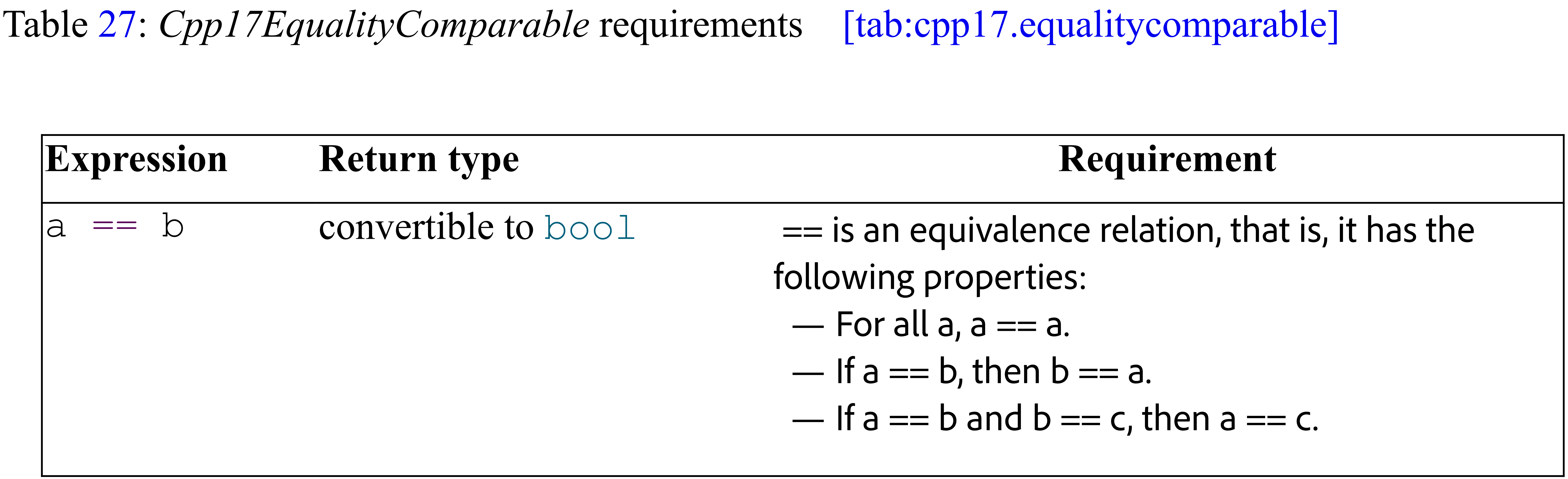


Table 27: *Cpp17EqualityComparable* requirements [tab:cpp17.equalitycomparable]

Expression	Return type	Requirement
`a == b`	convertible to `bool`	== is an equivalence relation, that is, it has the following properties: — For all a, a == a. — If a == b, then b == a. — If a == b and b == c, then a == c.

The term *domain of the operation* is used in the ordinary mathematical sense to denote the set of values over which an operation is (required to be) defined. This set can change over time. Each component may place additional requirements on the domain of an operation. These requirements can be inferred from the uses that a component makes of the operation and are generally constrained to those values accessible through the operation's arguments.

Domain of the Operation

The domain of an operation is *not* the types of the arguments

For a type, T, to satisfy a requirement, P:

$$\forall a \; P(a)$$

T must guarantee that

$$\exists a \in T \ni P(a)$$

`double` and `float` satisfy EqualityComparable

- So long as `nan` is not in the set being compared
- The absence of `nan` in the sequence for `find()` is a precondition of EqualityComparable

Weaker Preconditions

std::find_if() will return the first element for which a predicate is true

```
template <class I, class T>
I find(I first, I last, const T& value) {
    return find_if(first, last, [&](const auto& e) {
        return value == e;
    });
}
```

The additional requirements comes with the use of operator==

Otherwise the meaning of find and the meaning of equality is weakened

- Our ability to reason about code is weakened

Weaker Preconditions

std::find_if() will return the first element for which a predicate is true

```
template <class I, class T>
I find(I first, I last, const T& value) {
    return find_if(first, last, [&](const auto& e) {
        return e == value;
    });
}
```

The additional requirements comes with the use of operator==

Otherwise the meaning of find and the meaning of equality is weakened

- Our ability to reason about code is weakened

std::find() is broken in C++20

`std::find()` doesn't require that there exist any equality comparable values in `T`

`std::find()` doesn't guarantee that it finds `value`, even if `value` exists in the sequence

The meaning of `std::find()` is reduced to *works-as-implemented*

Fortunately, it is trivial to show that iff `operator==()` models *EqualityComparable*

- And all values in the sequence and the value being sought are in the *domain of* ==
- Then `std::find()` will *find*

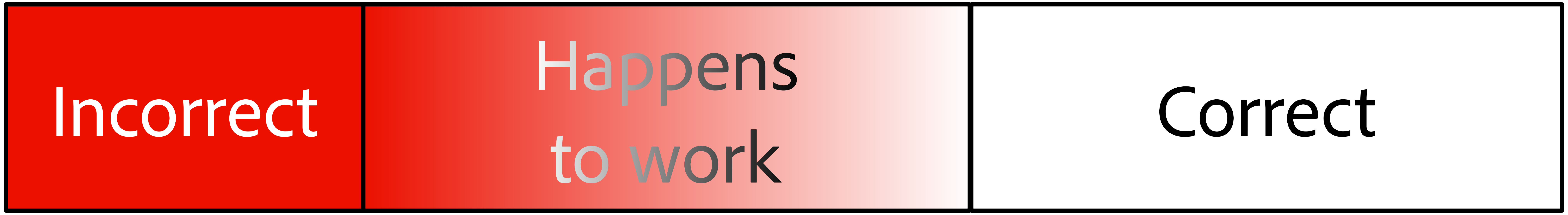
“Understanding why software fails is important, but the real challenge is understanding why software works.”

– Alexander Stepanov

Incorrect

Correct

Weakening Requirements



Incorrect

Happens to work

Correct

"I work with very good programmers and I see a ton of happens-to-work and very little actually correct."

– Titus Winters

What You Can Do

After trial-&-error, Stack Overflow, and Googling maybe you have code that happens-to-work

Take the time to read the specification

- Lookup anything you aren’t clear about
- Ask specific questions of experts

Clever uses require additional validation

What You Can Do

Think about the meaning, the semantics, of your code

- Ensure your use reflects the implied semantics
- Ensure your names reflect the semantics of what they represent

Specify your preconditions with a specification, concepts, and constraints

- Associate semantics with concepts
- `assert()` what you can, and what the compiler and tooling cannot check

Responsibility is multiplied for library writers and committee members

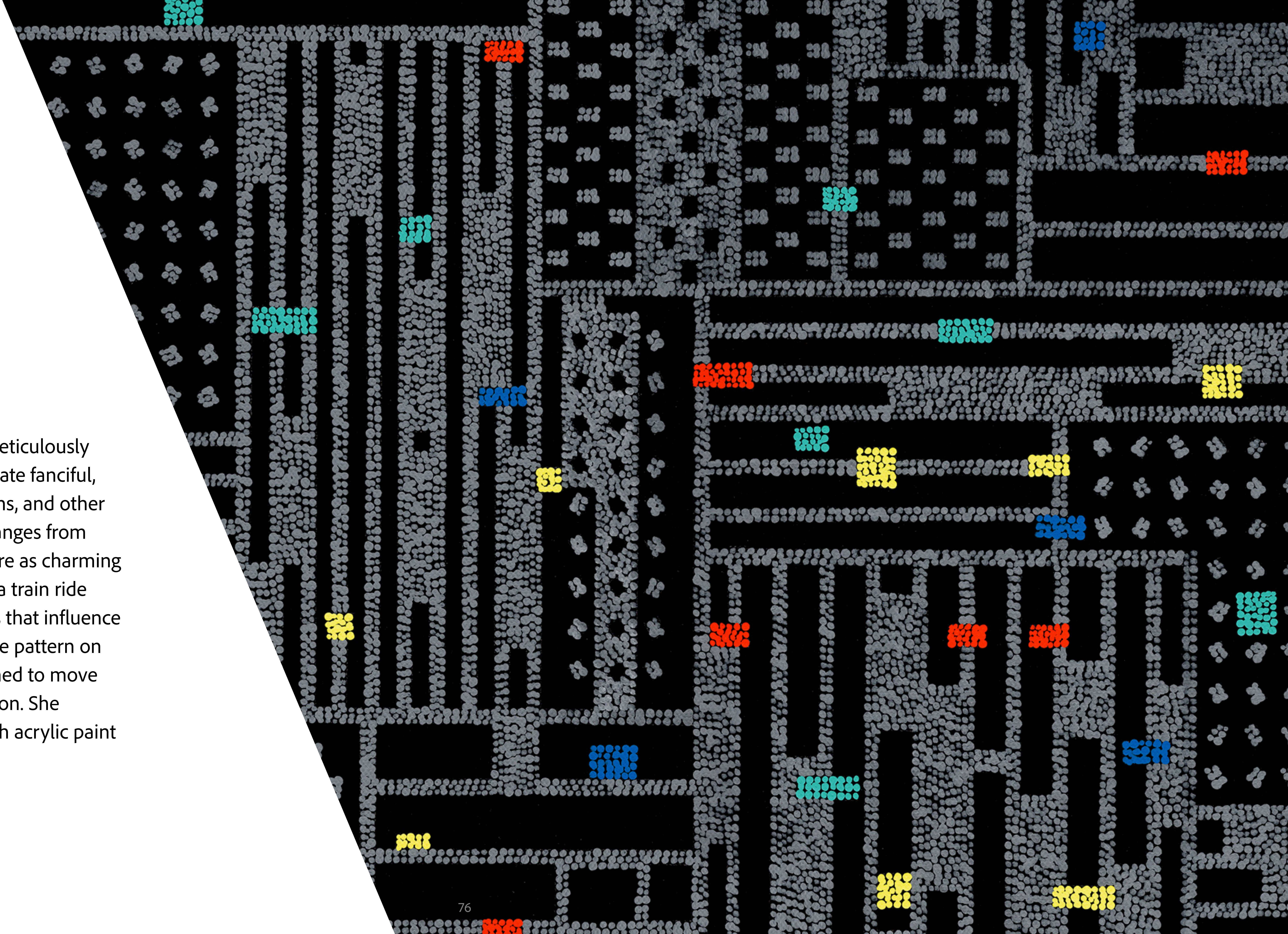
“The gap between code that fails and code that is correct is vast. Within it lies all the code that happens-to-work. Strive to write correct code and you will write better code.”

– Me, This Talk

Adobe

adobe.com/careers
bit.ly/adobecpp

About the artist



Momomi Sato

Tokyo-based artist Momomi Sato meticulously applies paint using toothpicks to create fanciful, pointillistic works of animals, patterns, and other colorful subjects. With a style that ranges from abstract to kawaii, Sato's paintings are as charming as they are beautiful. For this piece, a train ride prompted an exploration of systems that influence daily life. As she stared intently at the pattern on the seats, the lines and shapes seemed to move and draw Sato into another dimension. She recreated the sensation by hand with acrylic paint on canvas.

Adobe

Artwork by **Momomi Sato** / Japan