accu 2022

STRUCTURED CONCURRENCY

LUCIAN RADU TEODORESCU

Structured Concurrency

LUCIAN RADU TEODORESCU
GARMIN

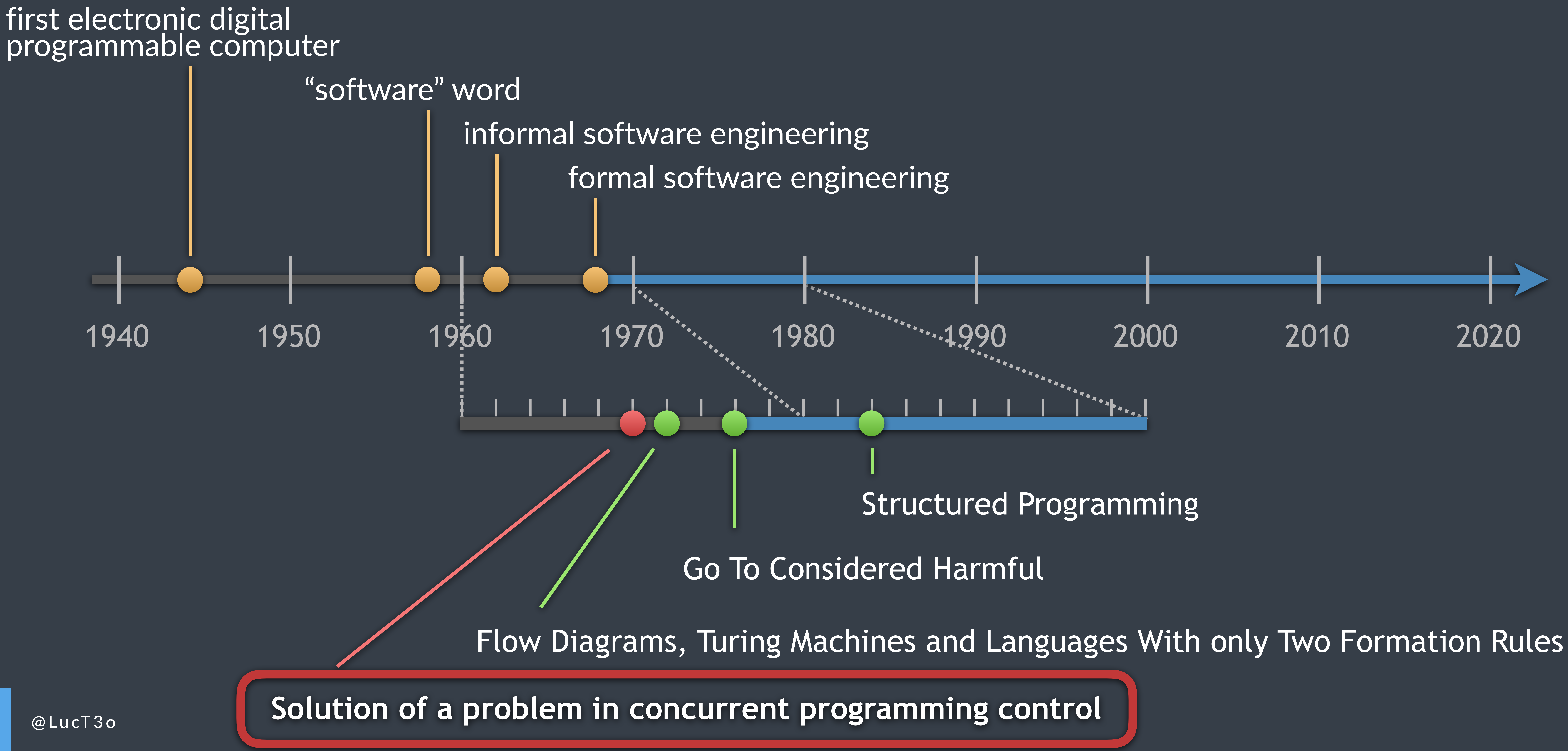
153

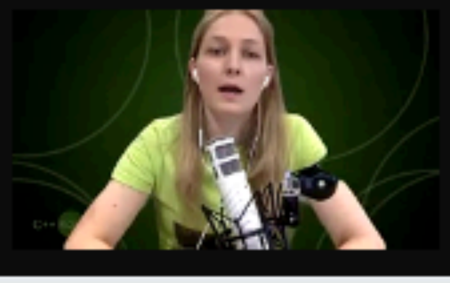
a tale of **two problems**

complexity

concurrency

relevant timeline

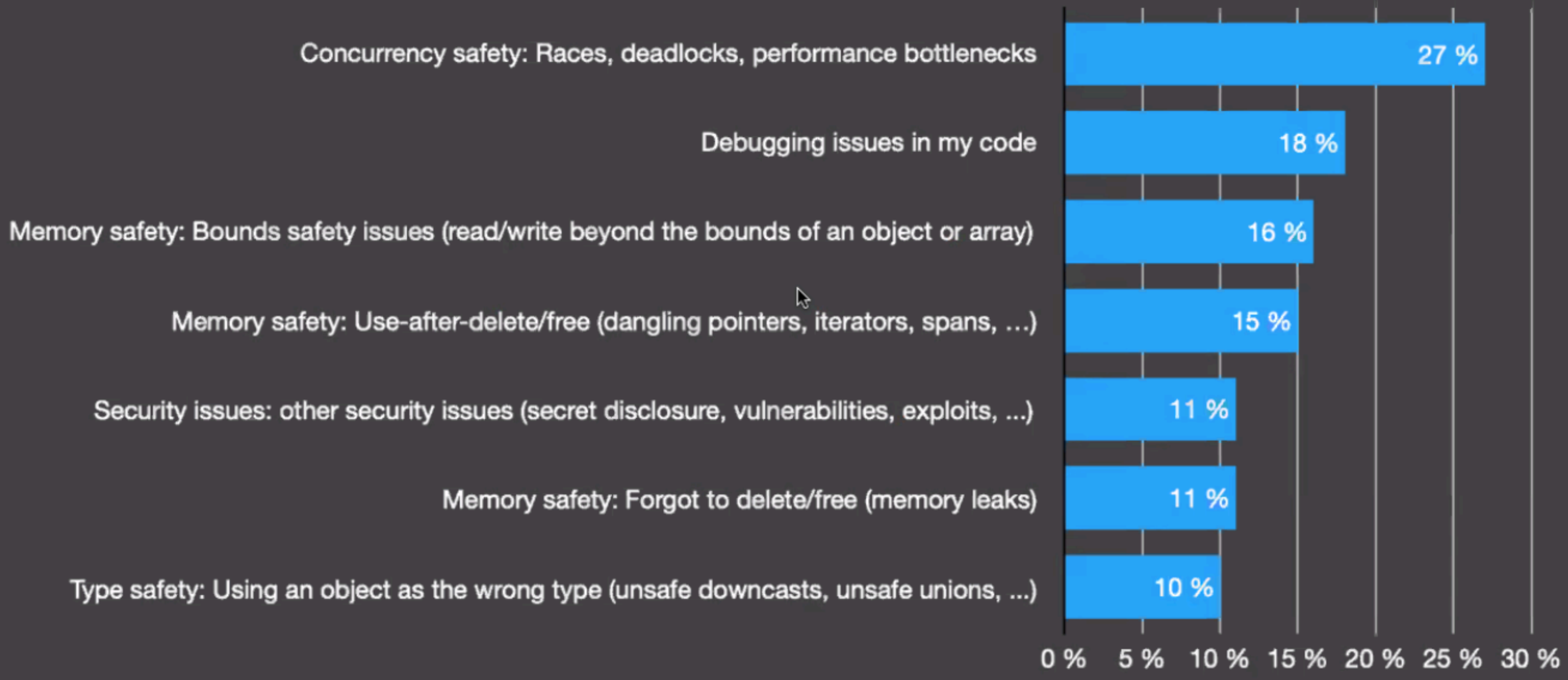




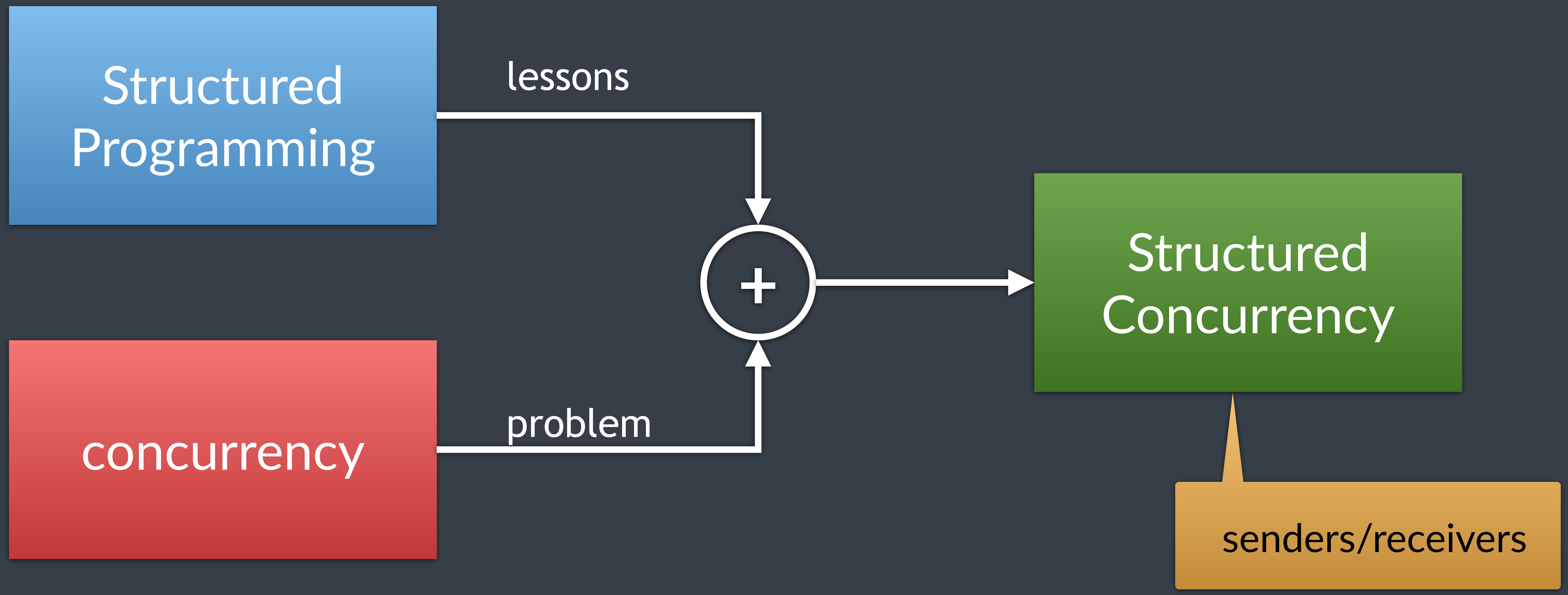
Code analysis++ EN - Google S

docs.google.com/presentation/d/1rfdksmy28o0Oc4EA1MnP2dQE-vyf7G4MDbZKPj0AAmU/edit#slide=id.p

C++ development frustration: safety & security



this talk


Structured Programming
lessons
concurrency
problem
+
Structured Concurrency
senders/receivers

what to expect

general picture

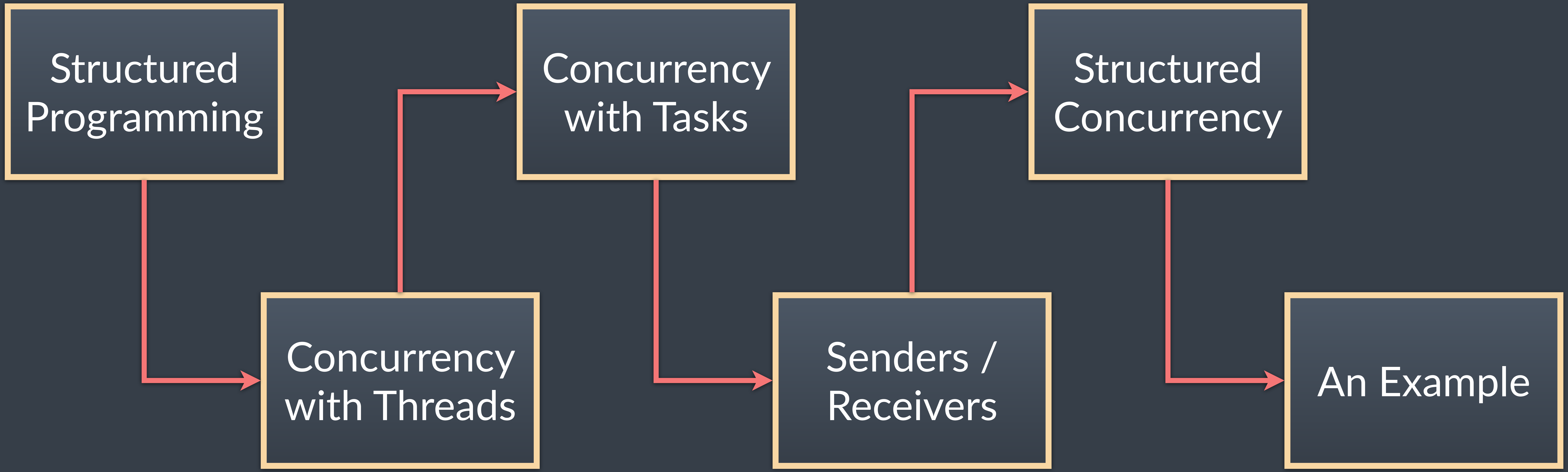
the **why**

what **NOT** to expect

introduction to senders/receivers

deep dive into details

Agenda


Structured Programming
Concurrency with Threads
Concurrency with Tasks
Senders / Receivers
Structured Concurrency
An Example

Structured Programming

1

Computational Linguistics

D. G. BOBROW, Editor

Flow Diagrams, Turing Machines And Languages With Only Two Formation Rules

Corrado Böhm and Giuseppe Jacopini
International Computation Centre and Istituto Nazionale per le Applicazioni del Calcolo, Roma, Italy

In the first part of the paper, flow diagrams are introduced to represent *inter al.* mappings of a set into itself. Although not every diagram is decomposable into a finite numbei of given base diagrams, this becomes tiue at a semantical level due to a suitable extension of the given set and of the basic mappings defined in it. Two normalization methods of flow diagrams are given. The first has three base diagrams; the second, only two.

In the second part of the paper, the second method is applied to the theory of Turing machines. With every Turing maching provided with a two-way half-tape, there is associated a similar machine, doing essentially the same job, but working on a tape obtained from the first one by interspersing alternate blank squares. The new machine belongs to the family, elsewhere introduced, generated by composition and iteration from the two machines λ and *R*. That family is a proper subfamily of the whole family of Turing machines.

1. Introduction and Summary

The set of block or flow diagrams is a two-dimensional programming language, which was used at the beginning of automatic computing and which now still enjoys a certain favor. As far as is known, a systematic theory of this language does not exist. At the most, there are some papers by Peter [1], Gorn [2], Hermes [3], Ciampa [4], Riguet [5], Ianov [6], Asser [7], where flow diagrams are introduced with different purposes and defined in connection with the descriptions of algorithms or programs.

This paper was presented as an invited talk at the 1964 International Colloquium on Algebraic Linguistics and Automata Theory, Jerusalem, Israel. Preparation of the manuscript was supported by National Science Foundation Grant GP-2880.

This work was carried out at the Istituto Nazionale per le Applicazioni del Calcolo (INAC) in collaboration with the International Computation Centre (ICC), under the Italian Consiglio Nazionale delle Ricerche (CNR) Research Group No. 22 for 1963–64.

In this paper, flow diagrams are introduced by the ostensive method; this is done to avoid definitions which certainly would not be of much use. In the first part (written by G. Jacopini), methods of normalization of diagrams are studied, which allow them to be decomposed into base diagrams of three types (first result) or of two types (second result). In the second part of the paper (by C. Böhm), some results of a previous paper are reported [8] and the results of the first part of this paper are then used to prove that every Turing machine is reducible into, or in a determined sense is equivalent to, a program written in a language which admits as formation rules only composition and iteration.

2. Normalization of Flow Diagrams

It is a well-known fact that a flow daigram is suitable for representing programs, computers, Turing machines, etc. Diagrams are usually composed of boxes mutually connected by oriented lines. The boxes are of functional

Edgar Dijkstra: Go To Statement Considered Harmful

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

Editor:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while** *B* **repeat** *A* or **repeat** *A* **until** *B*). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called "dynamic index," inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not.

APIC Studies in Data Processing No. 8

STRUCTURED PROGRAMMING

O.-J. Dahl, E. W. Dijkstra and C. A. R. Hoare

Academic Press
London New York San Francisco
A Subsidiary of Harcourt Brace Jovanovich, Publishers

@LucT3o

what is **Structured Programming**?

1. abstractions as building blocks
2. recursive decomposition
3. local reasoning
4. single entry, single exit point
5. soundness and completeness

1. abstractions as building blocks


Vancouver
Calgary
Edmonton
Seattle
Portland
Toronto
Ottawa
Montreal
San Francisco
Chicago
Minneapolis
Sacramento
Detroit
Buffalo
Cleveland
San José
Salt Lake City
Los Angeles
Las Vegas
San Diego
Denver
Phoenix
Houston
Monterrey
Guadalajara
Mexico City
Boston
New York
Newark
Philadelphia
Pittsburgh
Baltimore
St. Louis
Washington
Dallas
Atlanta
Jacksonville
Miami
Santo Domingo
San Juan
Lisbon
Seville
Alicante
Malaga
Valencia
Caracas
Recife
Fortaleza
Salvador
Rio de Janeiro
Maracaibo
Medellin
Lima
Brasilia
Valparaíso
Goiânia
Belo Horizonte
Santiago
Sao Paulo
Porto Alegre
Montevideo
Buenos Aires
Edinburgh
Glasgow
Newcastle
Sheffield
Liverpool
Dublin
Manchester
Nottingham
Birmingham
London
The Hague
Rotterdam
Amsterdam
Antwerp
Brussels
Laon
Charleroi
Bielefeld
Berlin
Orleans
Paris
Lille
Rhine/Ruhr
Cologne
Bonn
Rouen
Rennes
Nantes
Bordeaux
Strasbourg
Ludwigshafen
Zurich
Karlsruhe
Lausanne
Stuttgart
Frankfurt
Munich
Nuremburg
Linz
Brescia
Toulouse
Montpellier
Lyon
Milan
Bologna
Porto
Bilbao
Grenoble
Turin
Genoa
Parma
Madrid
Barcelona
Palma de Mallorca
Marseille
Nice
Rome
Naples
Salerno
Palermo
Catania
Tunis
Algiers
Alexandria
Cairo
Under Construction
Lagos
Oslo
Gothenburg
Stockholm
Copenhagen
Hamburg
Hanover
Helsinki
Gdansk
Poznan
Cracow
Warsaw
St. Petersburg
Moscow
Minsk
Prague
Kiev
Dnipropetrovs'k
Vienna
Budapest
Kryvy Rih
Kharkiv
Donets'k
Odessa
Bucharest
Belgrade
Sofia
Istanbul
Bursa
Ankara
Thessaloniki
Athens
Izmir
Adana
Haifa
Tel Aviv
Jerusalem
Dubai
Tehran
Esfahan
Baghdad
Shiraz
Mashhad
Delhi
Kolkota
Mumbai
Bangalore
Chennai
Kazan
Perm
Yekaterinburg
Chelyabinsk
Nizhniy Novgorod
Ufa
Samara
Volgograd
Tbilisi
Yerevan
Baku
Tashkent
Almaty
Omsk
Novosibirsk
Krasnoyarsk
Chengdu
Chongqing
Guangzhou
Hanoi
Chaing Mai
Ho Chi Minh
Bangkok
Kuala Lumpur
Singapore
Melbourne
Sydney
Sapporo
Sendai
Tokyo
Changchun
Kawasaki
Yokohama
Nagoya
Shenyang
Kyoto
Osaka
Beijing
Kobe
Hiroshima
Tianjin
Kitakyushu
Fukuoka
Pyongyang
Seoul
Xi'an
Naha
Incheon
Daejeon
Wuhan
Daegu
Nanjing
Gwangju
Shanghai
Busan
Hangzhou
Taipei
Kaohsiung
Shenzhen
Hong Kong
Manila
Jakarta
Auckland

abstraction

keep **essential**

drop **accidental**

example: variable

keep: type, single value

drop: the current value

```
int i = 0;
while ( a[i] <= 10 )
    i++;
print(i);
```

example: **function**

keep: signature, entry/exit semantics, main idea
drop: implementation details

example: **function**

keep: signature, entry/exit semantics, main idea
drop: implementation details

name: summarisation of semantics


Plato’s man: featherless biped

Plato's man: **featherless biped**

abstraction helps our **mind**

focus on **essential**

in Structured Programming

functions

data structures

2. recursive decomposition

divide et impera

building programs

recursively decompose programs into parts

make one decision at a time (local context)

later decisions don't influence prev. decisions

successive refinement

example

print the first 1000 prime numbers

step 0

```
auto main() -> int {
    print_first_1000_prime_numbers();
    return 0;
}
```

step 1

```
auto print_first_1000_prime_numbers() -> void {
    // TODO: define array p of 1000 elements
    // TODO: fill array p with first 1000 prime numbers
    // TODO: print array p
}
```

step 2 (refinement)

```
auto print_first_1000_prime_numbers() -> void {
    int p[1000];
    fill_with_first_1000_prime_numbers(p);
    print_array(p);
}
```

step 3

```
auto fill_with_first_1000_prime_numbers(int* p) -> void {
    int num_primes = 0;
    int val = 1;
    while (num_primes < 1000) {
        // TODO: increase val until next prime number
        p[num_primes++] = val;
    }
}
```

step 4

```
auto fill_with_first_1000_prime_numbers(int* p) -> void {
    int num_primes = 0;
    int val = 1;
    while (num_primes < 1000) {
        do {
            val++; // WARNING: inefficient
        } while (!is_prime(val, p, num_primes));
        p[num_primes++] = val;
    }
}
```

step 5 (revision)

```
auto fill_with_first_1000_prime_numbers(int* p) -> void {
    p[0] = 2;
    int num_primes = 1;
    int val = 1;
    while (num_primes < 1000) {
        do {
            val += 2;
        } while (!is_prime(val, p, num_primes));
        p[num_primes++] = val;
    }
}
```

step 6

```
auto is_prime(int val, int* p, int num_primes) -> bool {
    // TODO: for all previously found primes, starting with 3, up until sqrt(val)
    //      ... check if there is a divisor of val
}
```

step 7 (refinement)

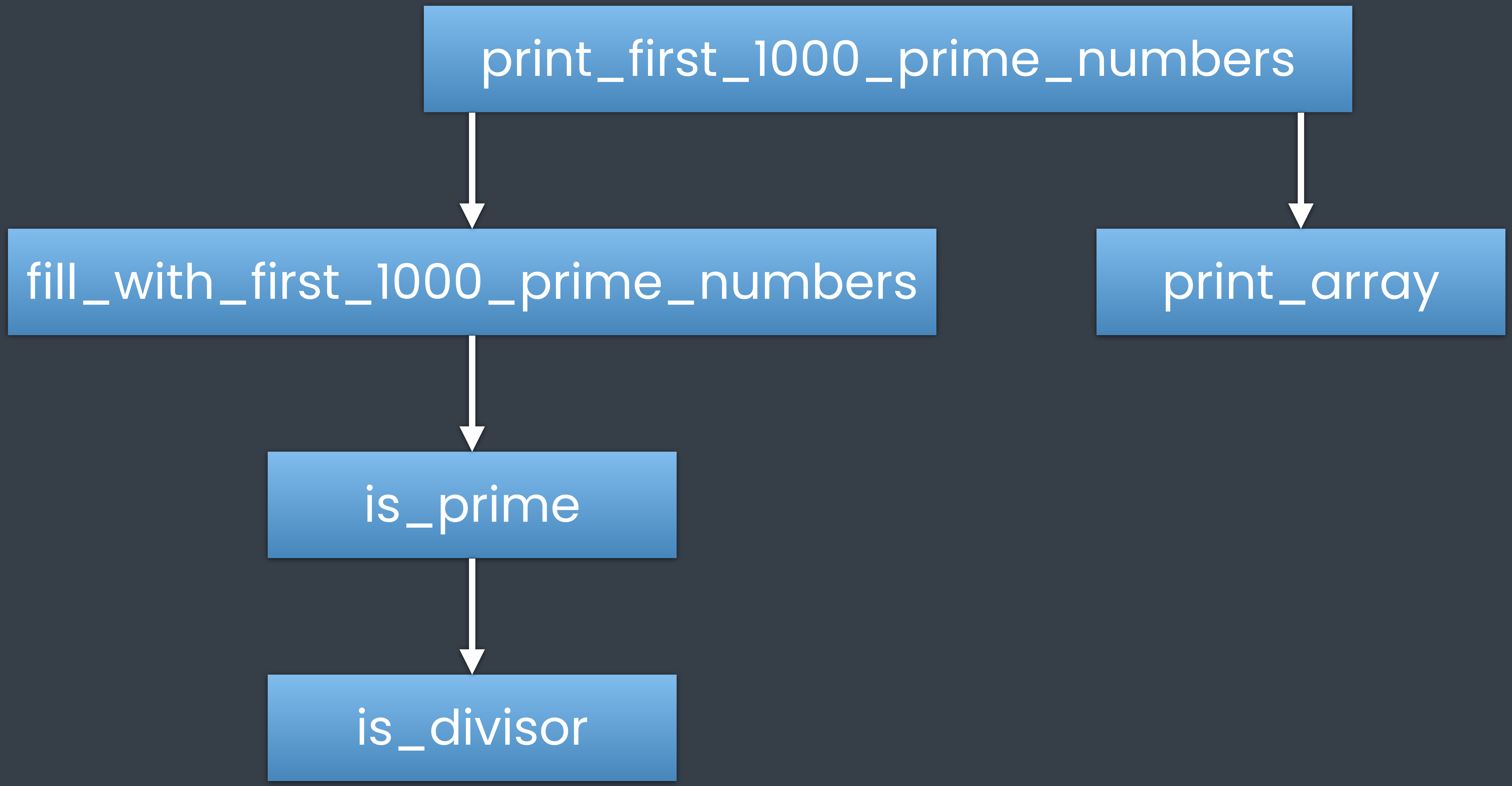
```
auto is_prime(int val, int* p, int num_primes) -> bool {
    bool val_is_prime = true;
    for (int i = 1; i < num_primes && p[i] * p[i] <= val && val_is_prime; i++) {
        val_is_prime = !is_divisor(p[i], val);
    }
    return val_is_prime;
}
```

step 8

```
auto is_divisor(int div, int val) -> bool {
    return (val % div) == 0;
}
```

step 9

```
auto print_array(int* p) -> void {
    for (int i = 0; i < 1000; i++)
        printf("%d\n", p[i]);
}
```


print_first_1000_prime_numbers
fill_with_first_1000_prime_numbers
print_array
is_prime
is_divisor

3. local reasoning

nested scopes

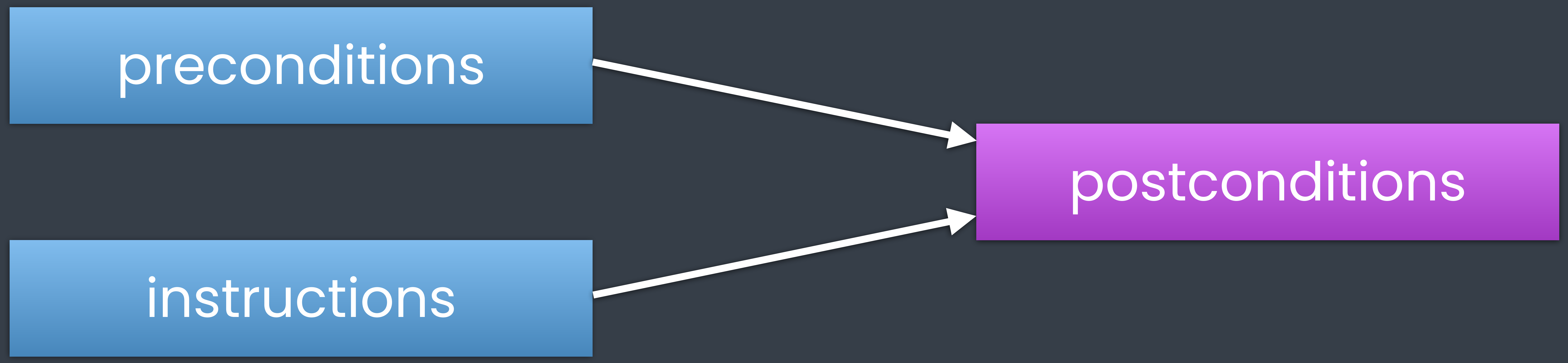
encapsulation of local concerns

small example

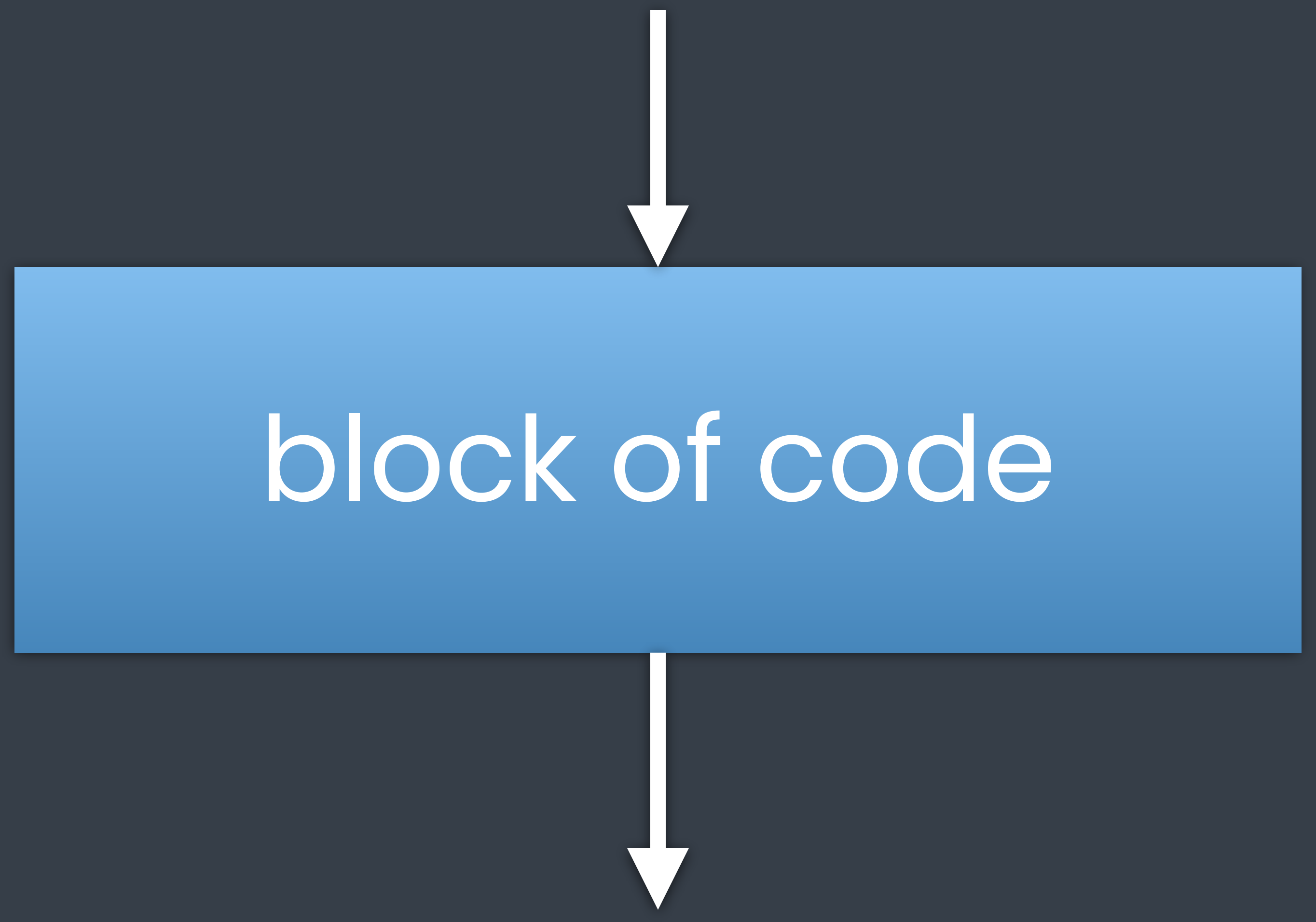
```
for (int i=0; i<10; i++) {
    int x = i;
    {
        for (int i=0; i<10; i++) {
            int y = i;
            printf("%d/%d\n", x, y);
        }
    }
}
```

focus

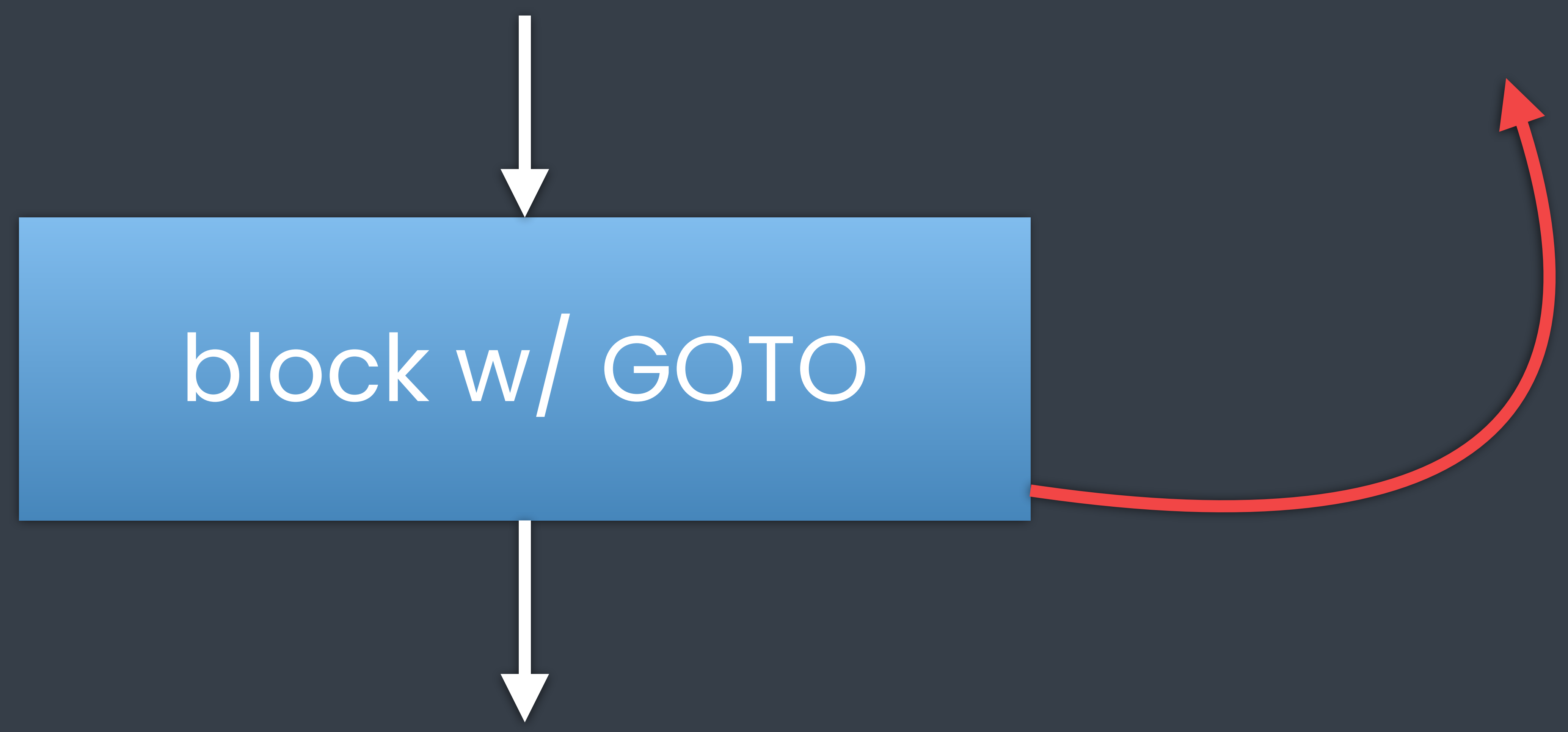
one thing at a time


preconditions
instructions
postconditions

4. single entry, single exit point



GOTO excluded



reasons?

easy

linear reasoning

code & execution have the same flow

same **shape**

instruction

function call

block of code

alternatives

loops

exit in C++

return a value

throw an exception

abort (internal or external)

```
auto sum(Matrix a, Matrix b) -> Matrix;

Matrix a = ...;
Matrix b = ...;
Matrix c = sum(a, b);
```

5. soundness and completeness

can Structured Programming be applied?

soundness

applying SP should lead to correct programs

completeness

applying SP for **all programs**

Böhm-Jacopini theorem

Computational Linguistics

D. G. BOBROW, Editor

Flow Diagrams, Turing Machines And Languages With Only Two Formation Rules

Corrado Böhm and Giuseppe Jacopini
International Computation Centre and Istituto Nazionale per le Applicazioni del Calcolo, Roma, Italy

In the first part of the paper, flow diagrams are introduced to represent *inter al.* mappings of a set into itself. Although not every diagram is decomposable into a finite numbei of given base diagrams, this becomes tiue at a semantical level due to a suitable extension of the given set and of the basic mappings defined in it. Two normalization methods of flow diagrams are given. The first has three base diagrams; the second, only two.

In the second part of the paper, the second method is applied to the theory of Turing machines. With every Turing maching provided with a two-way half-tape, there is associated a similar machine, doing essentially the same job, but working on a tape obtained from the first one by interspersing alternate blank squares. The new machine belongs to the family, elsewhere introduced, generated by composition and iteration from the two machines λ and R. That family is a proper subfamily of the whole family of Turing machines.

In this paper, flow diagrams are introduced by the ostensive method; this is done to avoid definitions which certainly would not be of much use. In the first part (written by G. Jacopini), methods of normalization of diagrams are studied, which allow them to be decomposed into base diagrams of three types (first result) or of two types (second result). In the second part of the paper (by C. Böhm), some results of a previous paper are reported [8] and the results of the first part of this paper are then used to prove that every Turing machine is reducible into, or in a determined sense is equivalent to, a program written in a language which admits as formation rules only composition and iteration.

2. Normalization of Flow Diagrams

It is a well-known fact that a flow daigram is suitable for representing programs, computers, Turing machines, etc. Diagrams are usually composed of boxes mutually connected by oriented lines. The boxes are of functional type (see Figure 1) when they represent elementary operations to be carried out on an unspecified object x of a set X, the former of which may be imagined concretely as the set of the digits contained in the memory of a computer, the tape configuration of a Turing machine, etc. There are other boxes of predicative type (see Figure 2) which do not operate on an object but decide on the next operation to be carried out, according to whether or not a certain property of $x \in X$ occurs. Examples of

Structured Programming

1. abstractions as building blocks
2. recursive decomposition
3. local reasoning
4. single entry, single exit point
5. soundness and completeness

Concurrency with Threads

2

primitives

threads

locks (mutexes, semaphores, etc.)

1. abstractions as building blocks

threads and locks are not good abstractions

2. recursive decomposition

into what?

3. local reasoning

threads and locks have non-local effects
(by design)

4. single entry, single exit point

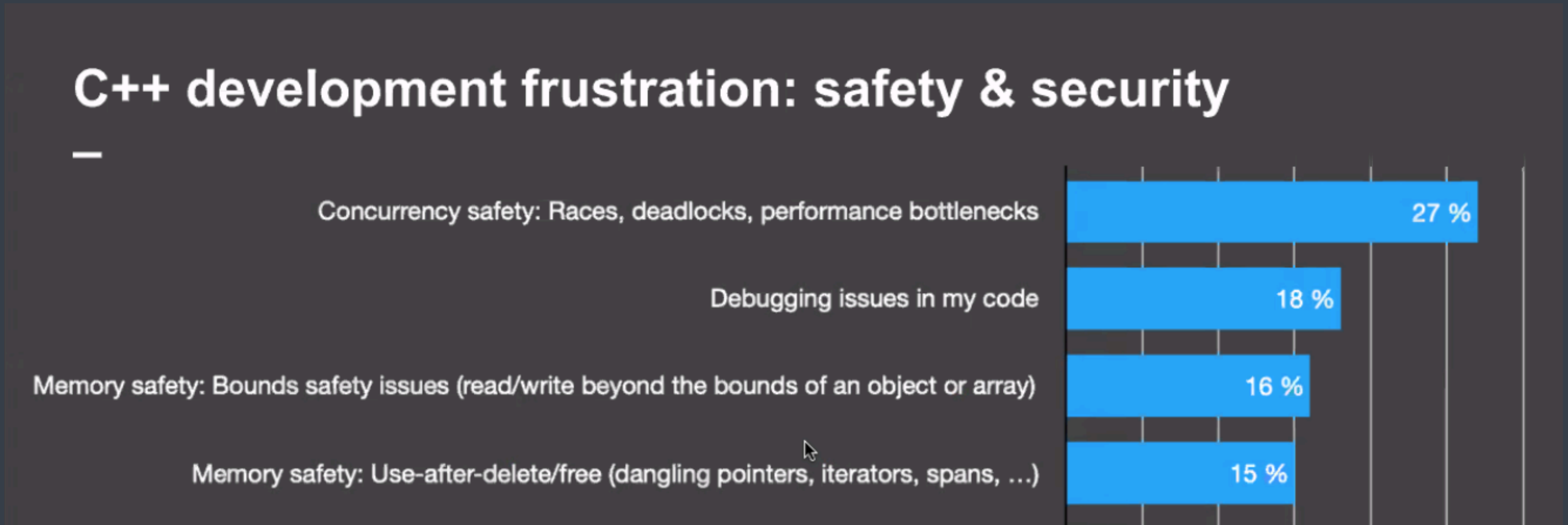
N/A

5. soundness and completeness

complete but not sound

un**sound**

no general strategy



structured?

no

~~abstractions as building blocks~~	no
~~recursive decomposition~~	no
~~local reasoning~~	no
~~single entry, single exit point~~	–
~~soundness~~ and completeness	½

Concurrency with Tasks

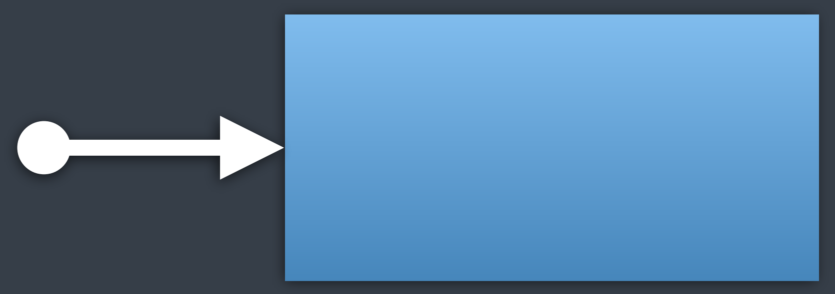
3

primitives

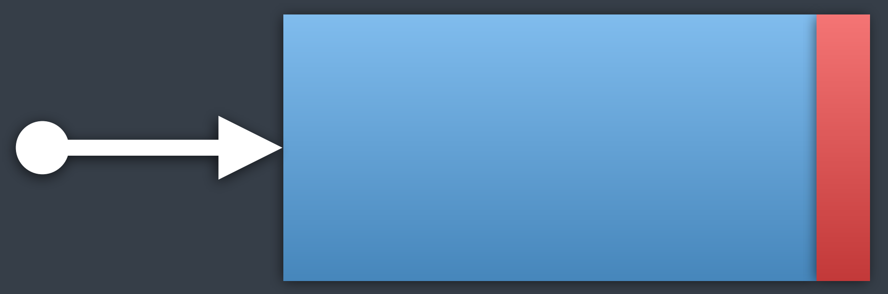
tasks
(independent units of work)

two models

raw tasks

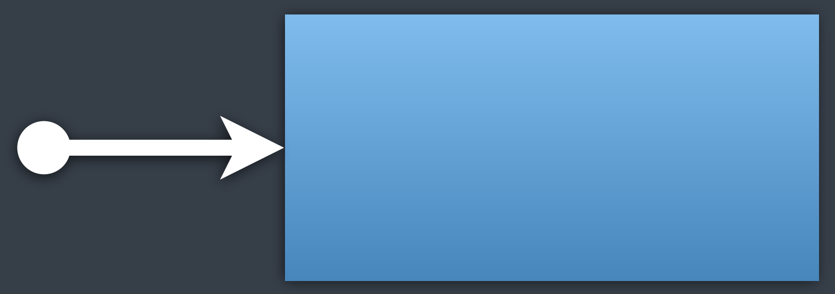


with continuations

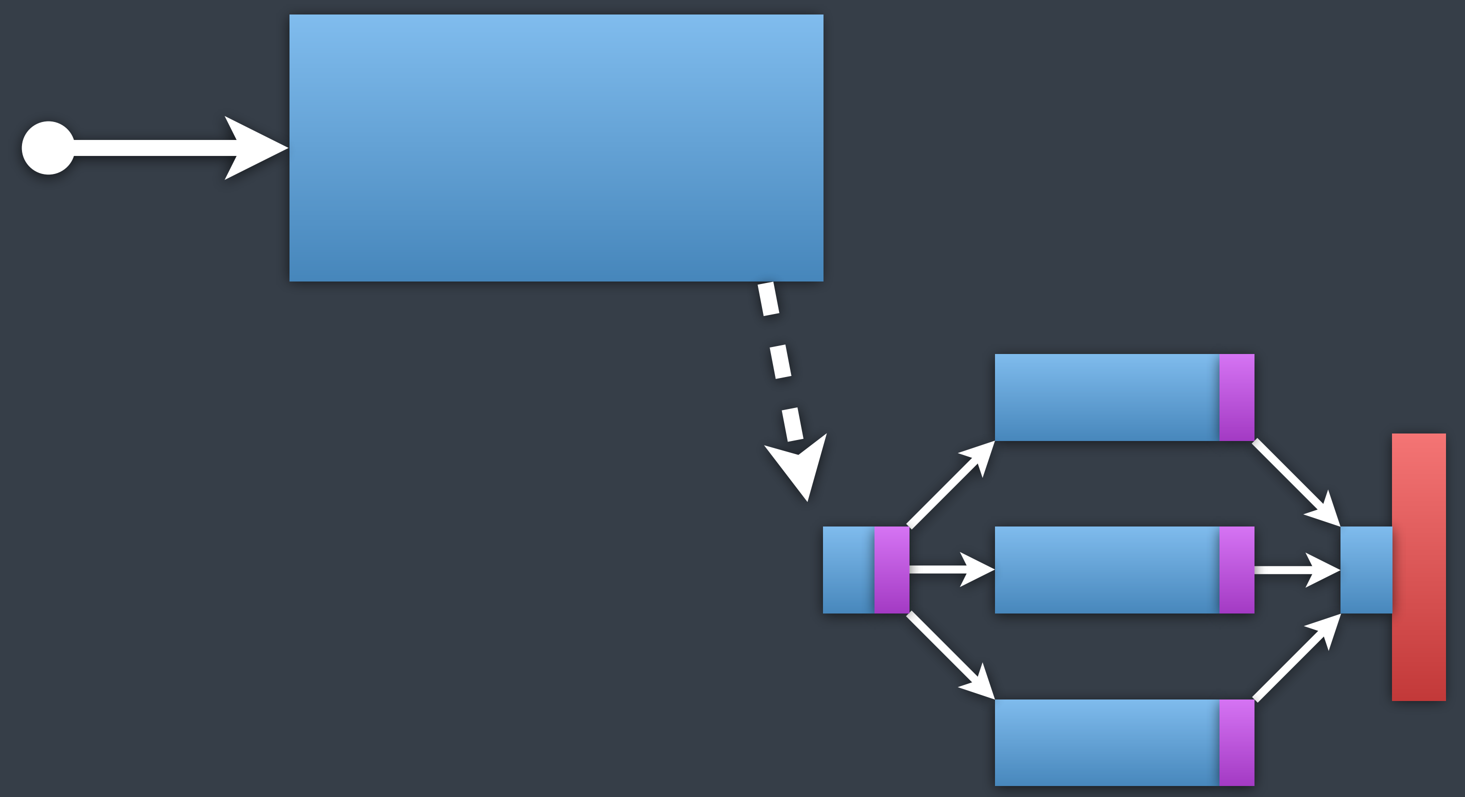


two models

raw tasks



with continuations



two models

raw tasks

just like functions

with continuations

can represent
larger concurrent parts

raw tasks — **structured**?

partial

abstractions as building blocks	yes
recursive decomposition	no
local reasoning	yes
single entry, single exit point	no
soundness and completeness	yes

tasks w/ cont. — **structured**?

partial

abstractions as building blocks	yes
recursive decomposition	yes
local reasoning	no
single entry, single exit point	½
soundness and completeness	yes

Senders/Receivers

4

P2300 — `std::execution`

C++ proposal

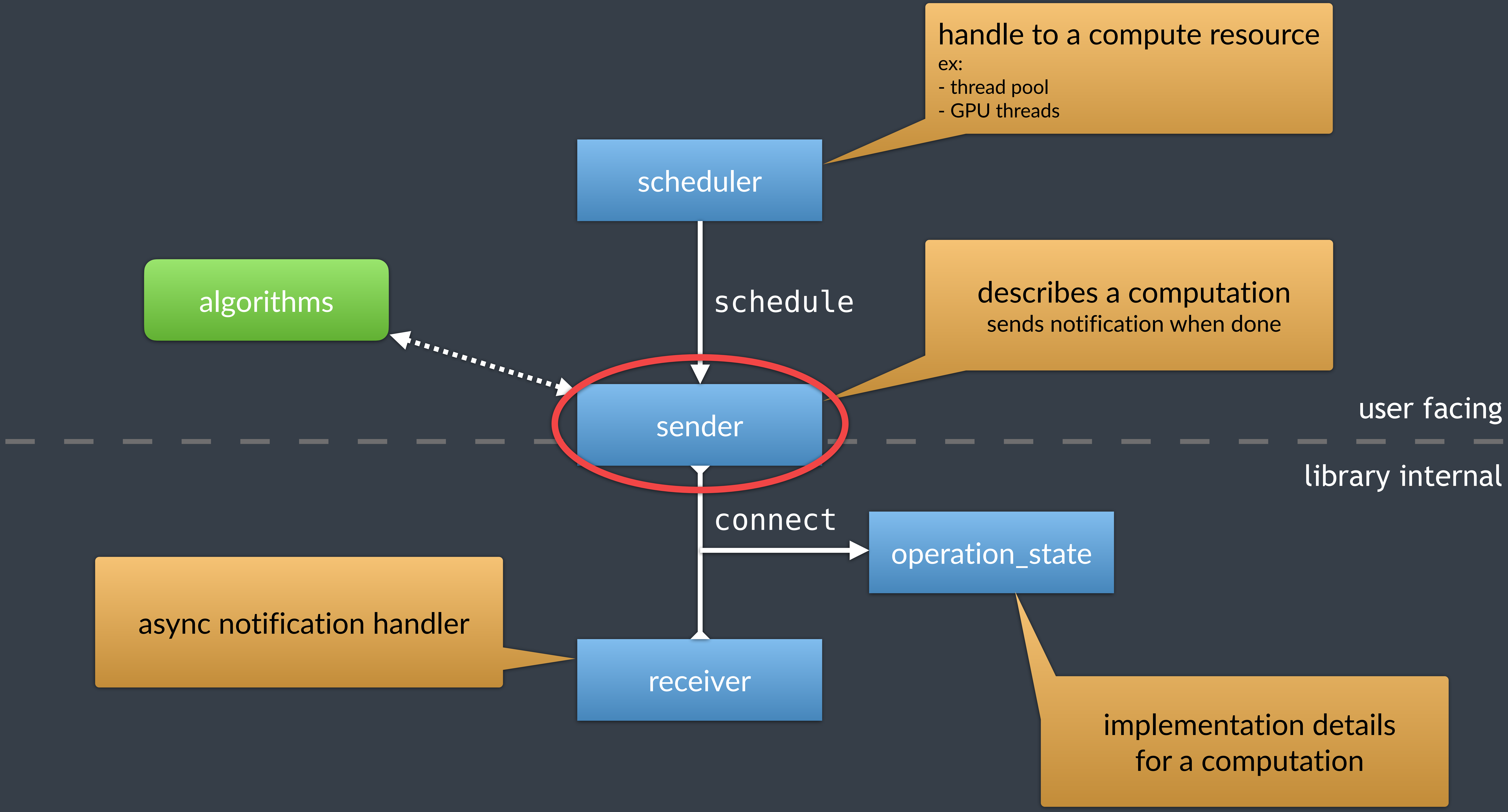
did not make it to C++23

P2300 — `std::execution`

concepts

initial set of algorithms

utilities

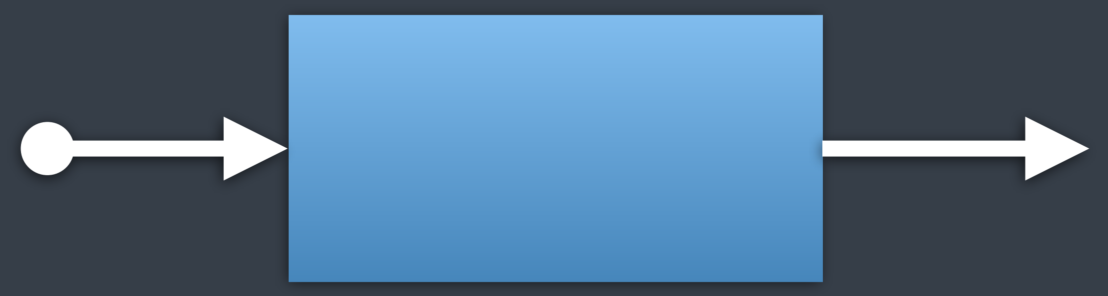

handle to a compute resource
ex:
- thread pool
- GPU threads
scheduler
schedule
algorithms
describes a computation
sends notification when done
sender
user facing
library internal
connect
operation_state
async notification handler
receiver
implementation details
for a computation

senders describe **computations**

any chunk of work,
with one entry and one exit point

computations

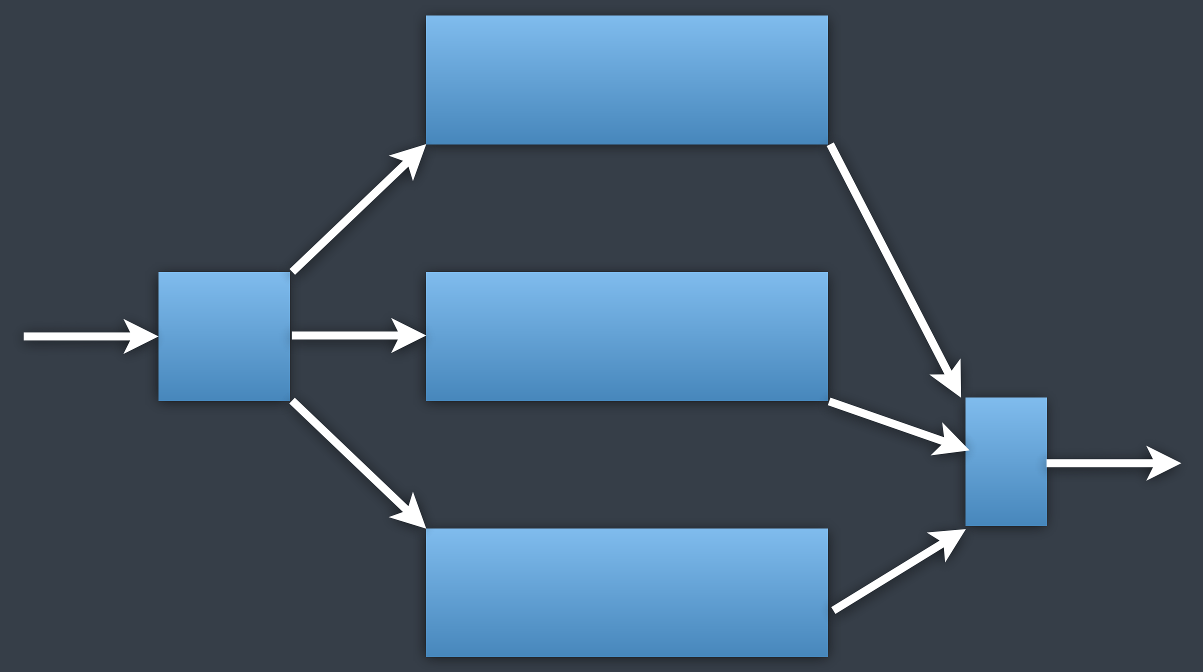
a task



computations

a task

tasks over multiple threads

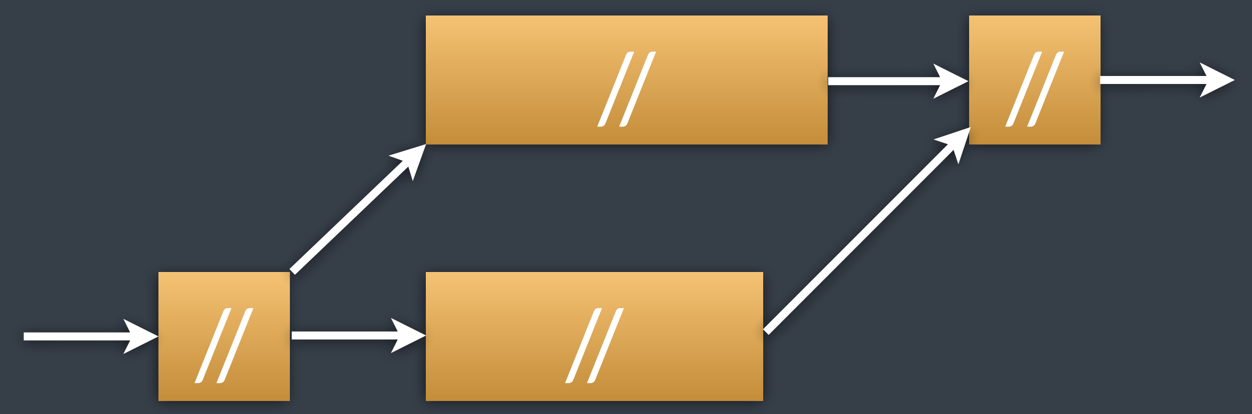


computations

a task

tasks over multiple threads

group of computations



computations

a task

tasks over multiple threads

group of computations

the entire application

computations

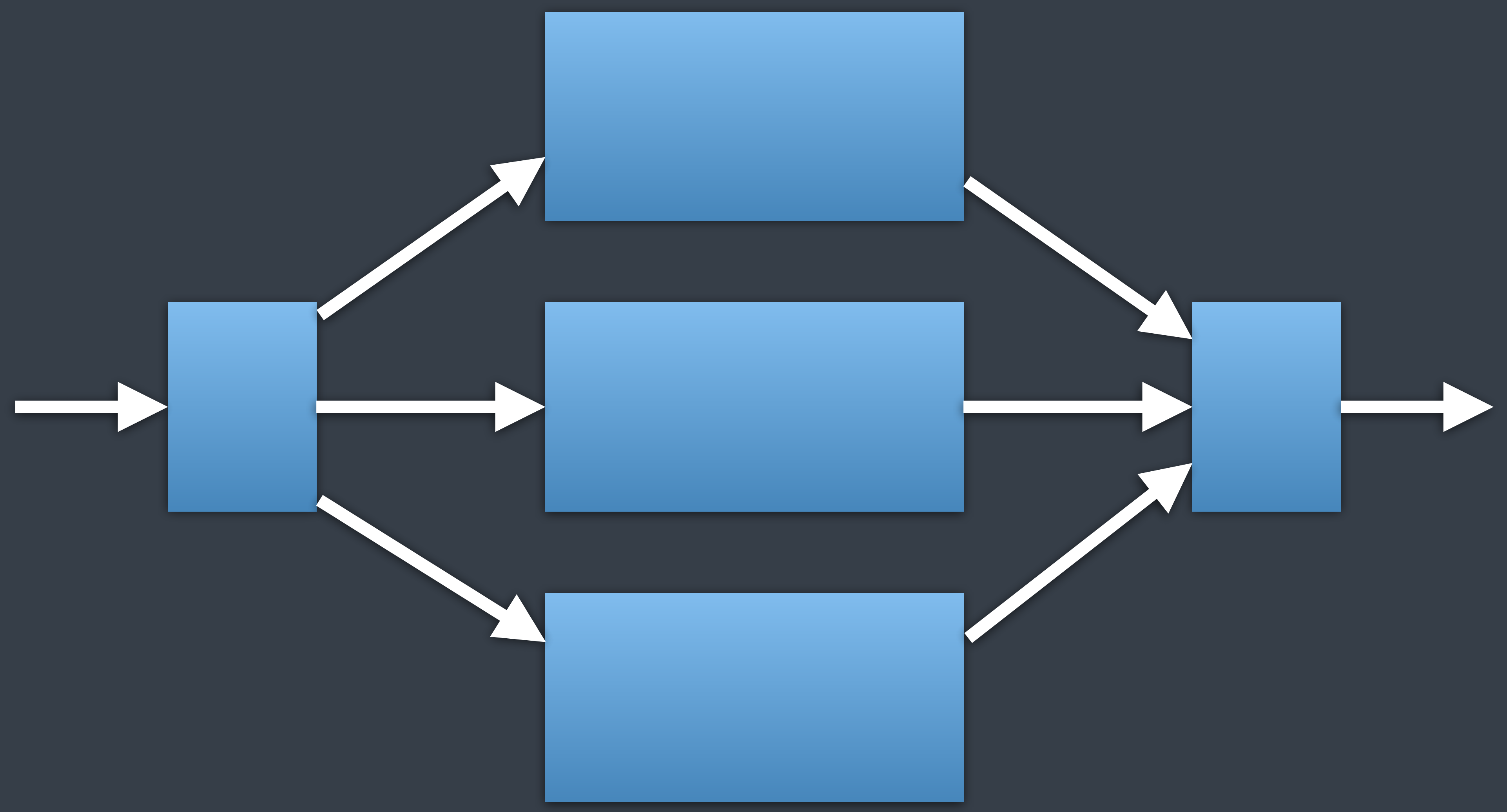
a task

tasks over multiple threads

group of computations

the entire application

example

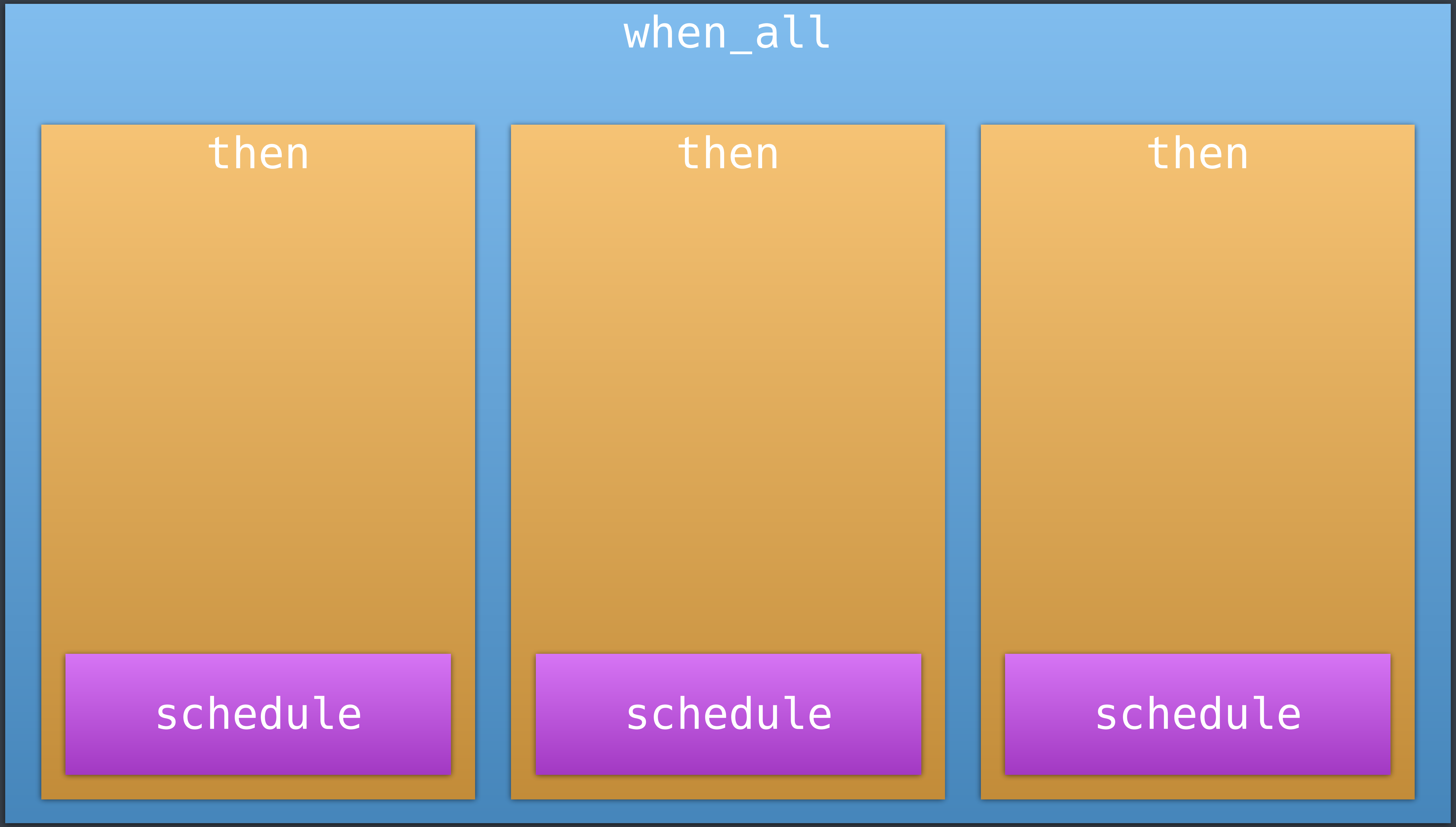


example

```
auto work1() -> int;
auto work2() -> double;
auto work3() -> std::string;
auto combine_res(int i, double d, const std::string& s) -> int;

auto compute_in_parallel() -> int {
    static_thread_pool pool{8};
    ex::scheduler auto sched = pool.get_scheduler();

    ex::sender auto work =
        ex::when_all(
            ex::schedule(sched) | ex::then(work1),
            ex::schedule(sched) | ex::then(work2),
            ex::schedule(sched) | ex::then(work3)
        );
    auto [i, d, s] = std::this_thread::sync_wait(std::move(work)).value();
    return combine_res(i, d, s);
}
```


when_all
then
then
then
schedule
schedule
schedule

senders's **completion**

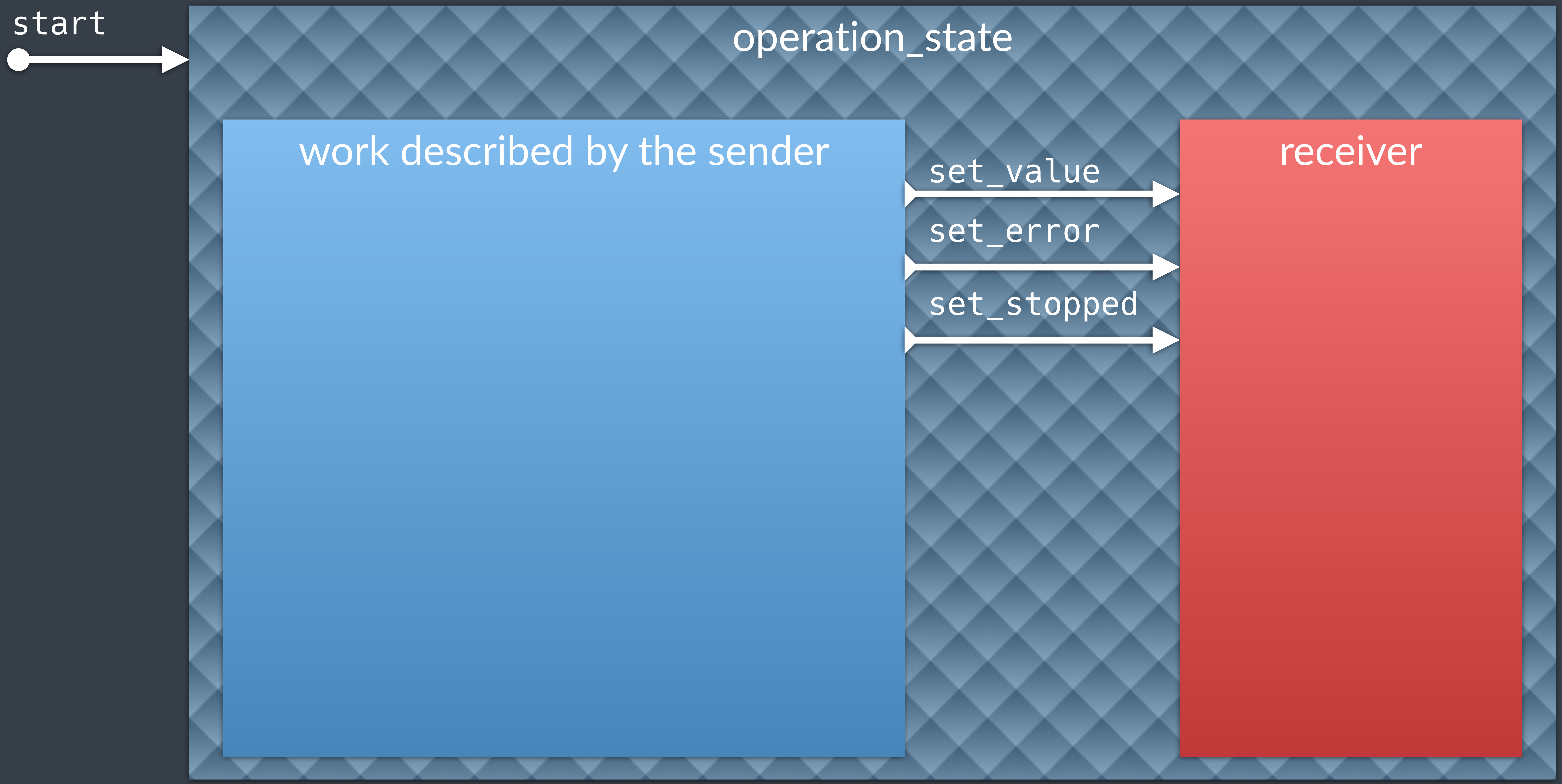
promises to call on completion, one of:

```
- set_value(dest, values…)
- set_error(dest, error)
- set_stopped(dest)
```

receiver

something that is called with one of:

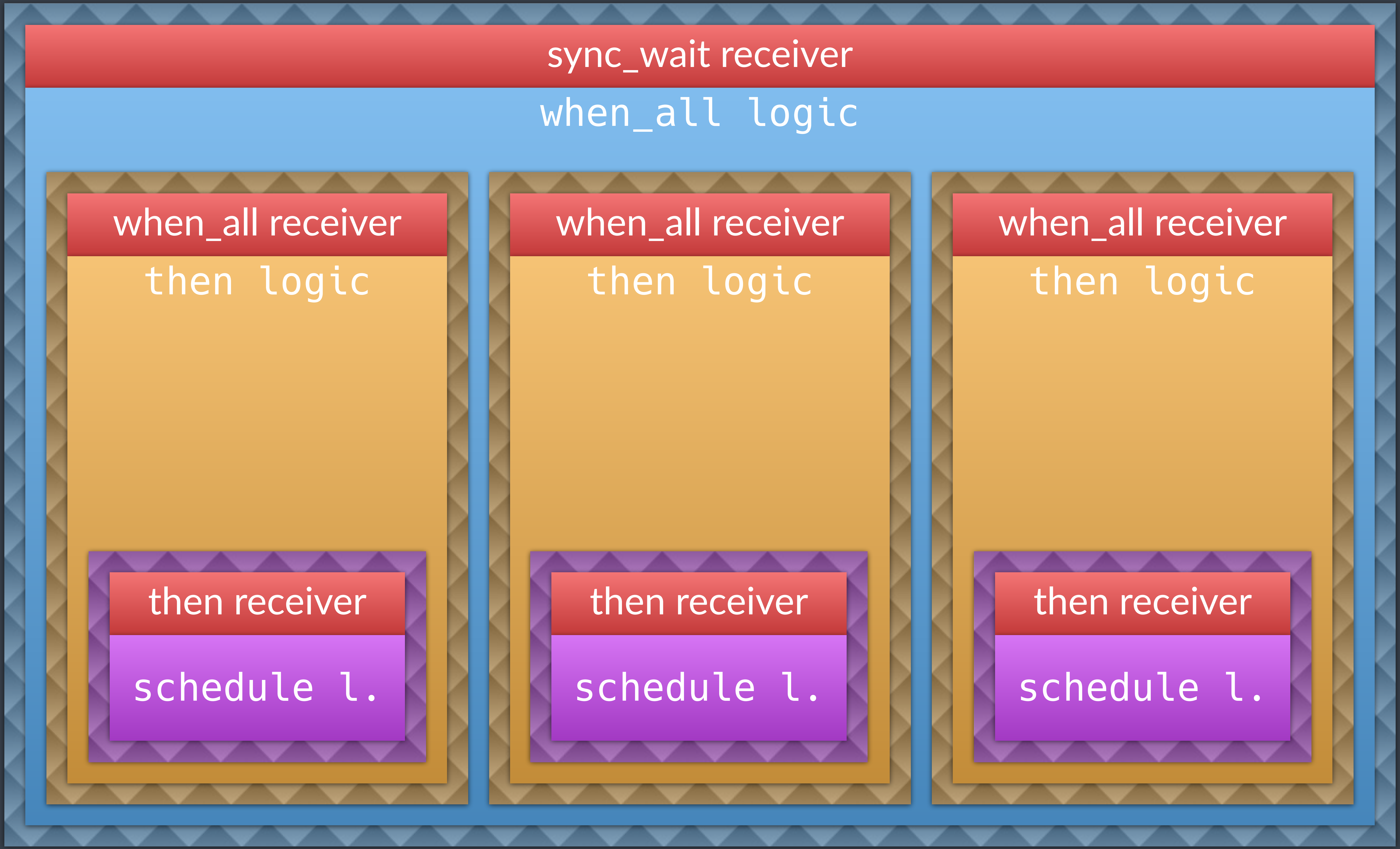
- `set_value(recv, *values…*)`
- `set_error(recv, *error*)`
- `set_stopped(recv)`


start
operation_state
work described by the sender
set_value
set_error
set_stopped
receiver

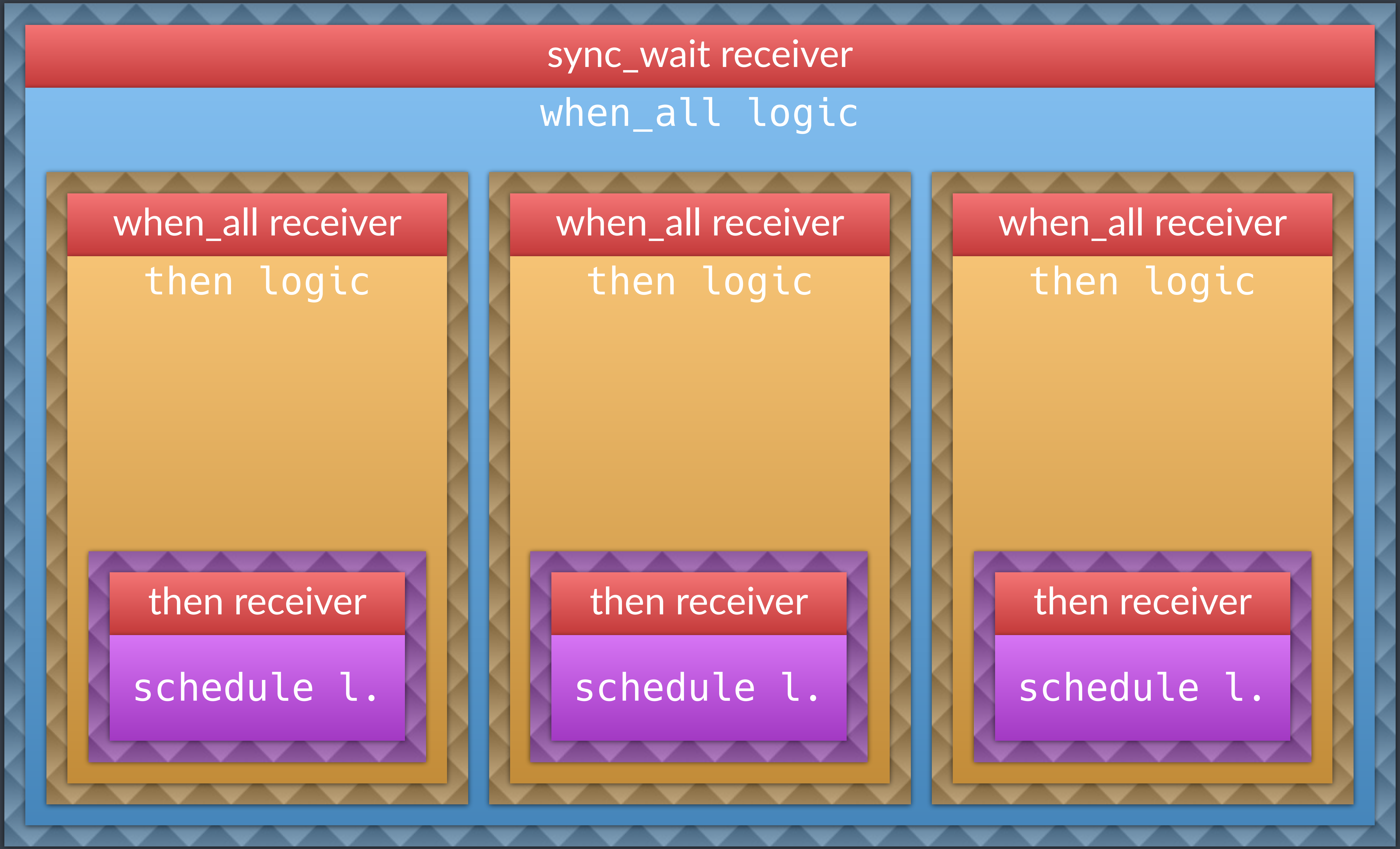
operation state object

alive during the whole duration

operation state creation

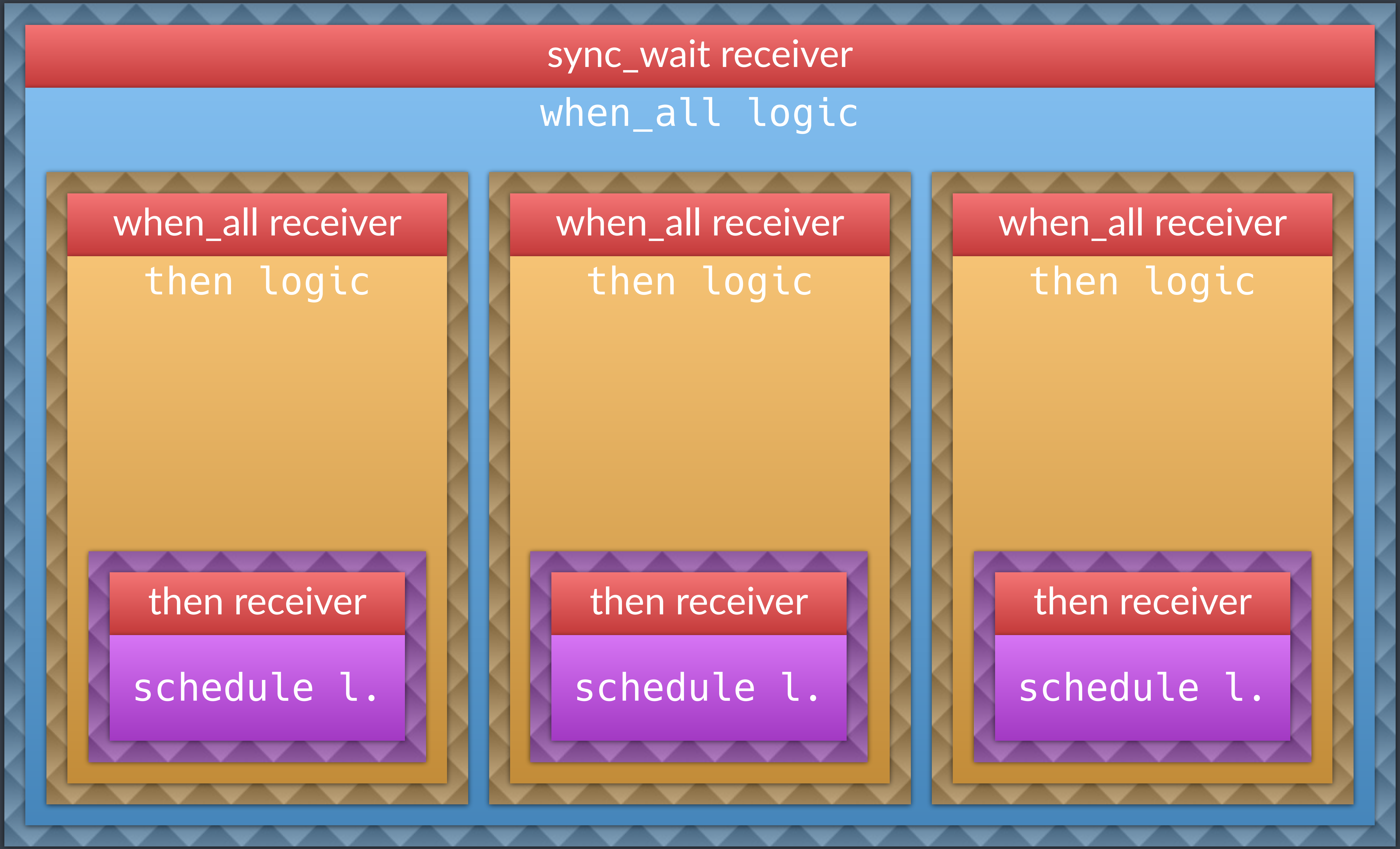


operation state creation

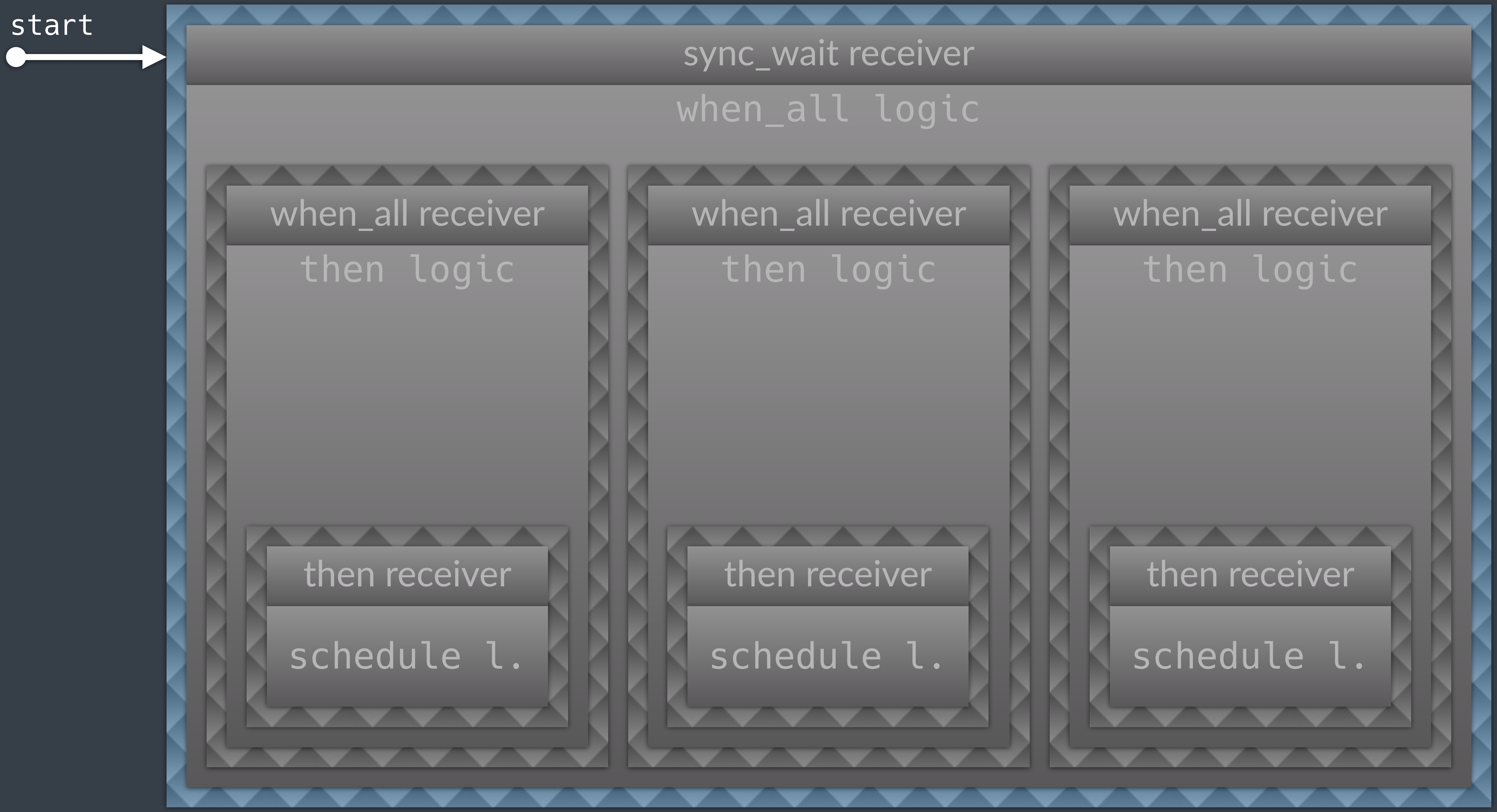


@LucT3o

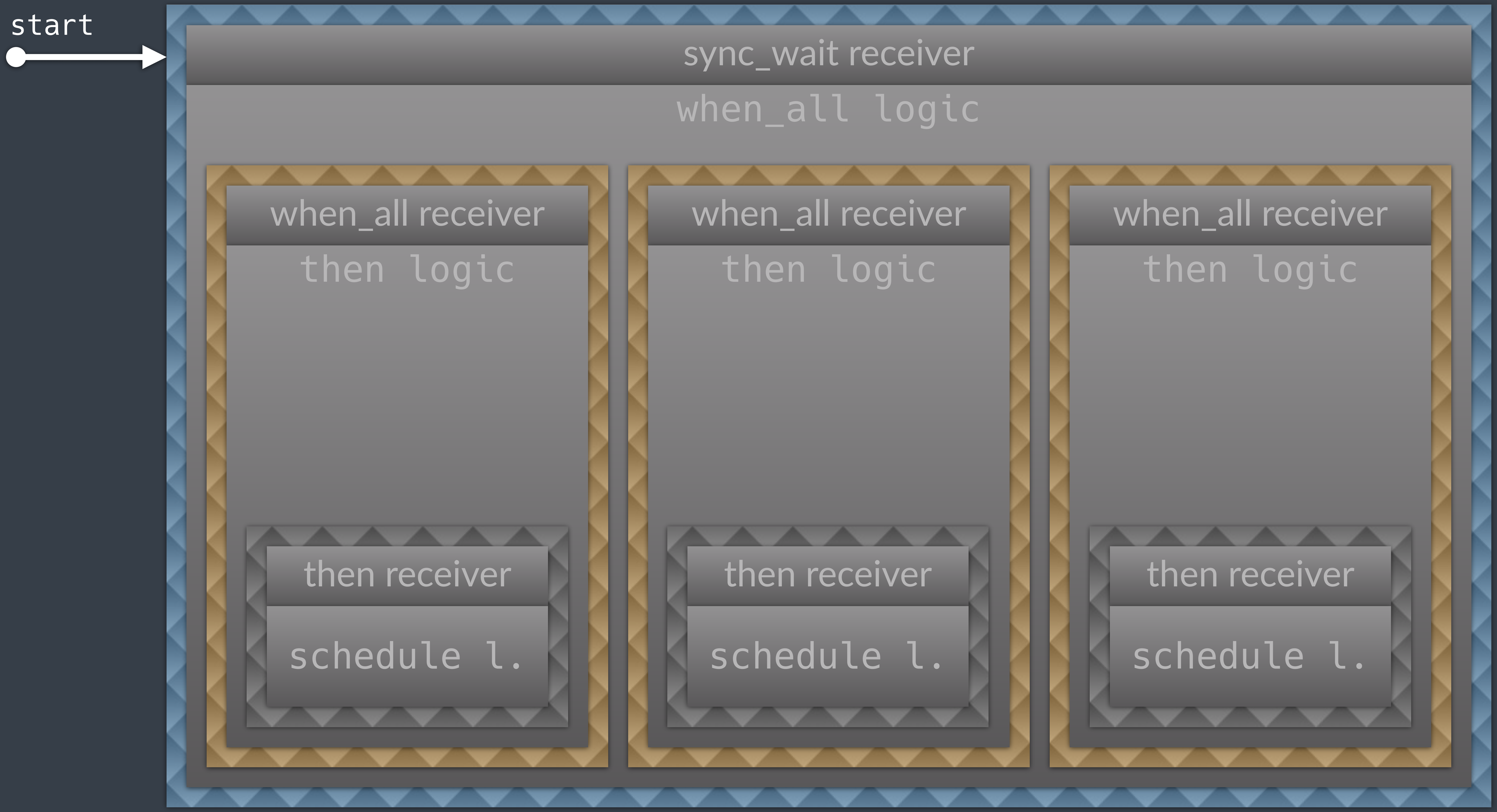
operation state creation



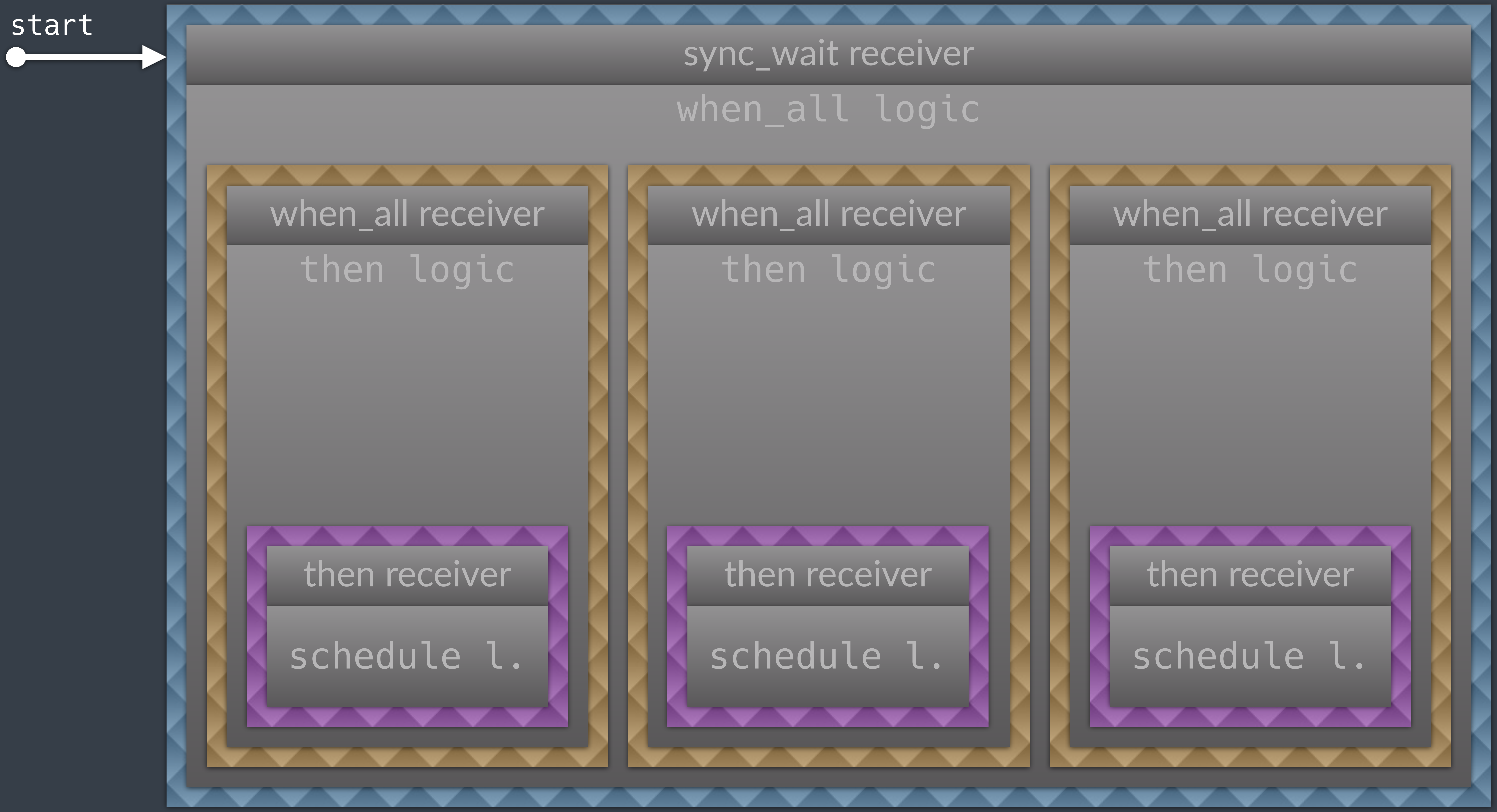
starting operation



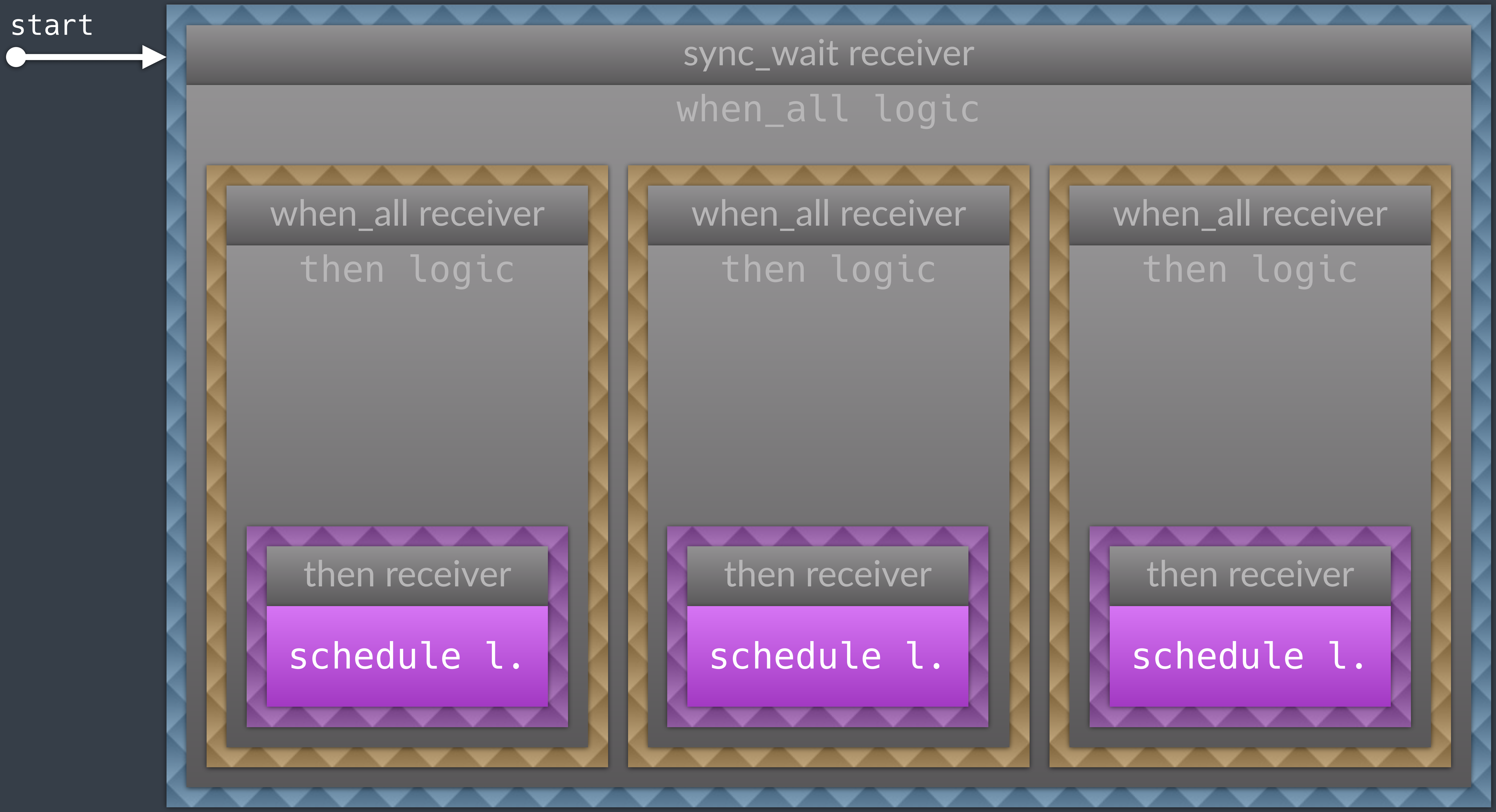
starting operation



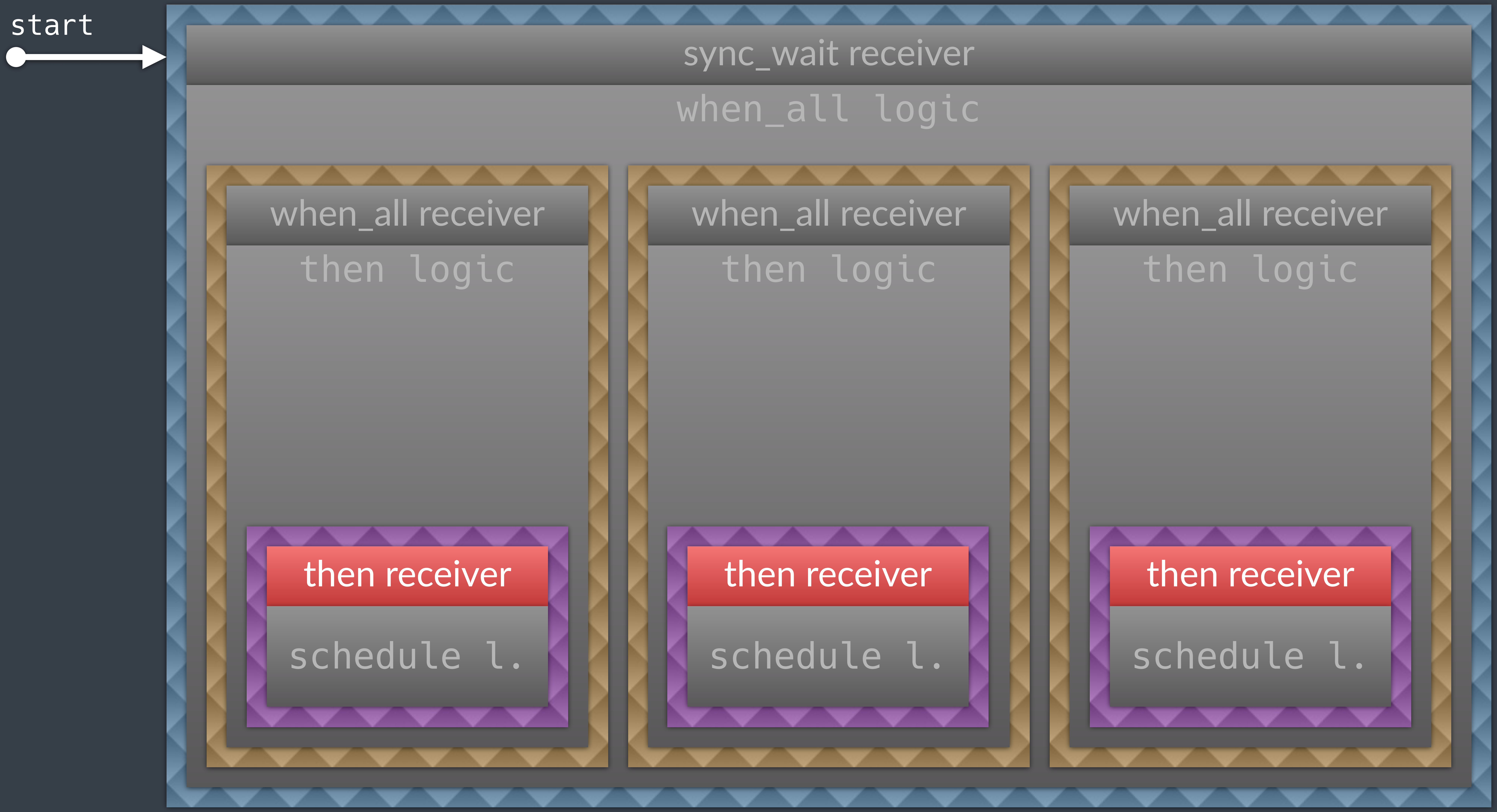
starting operation



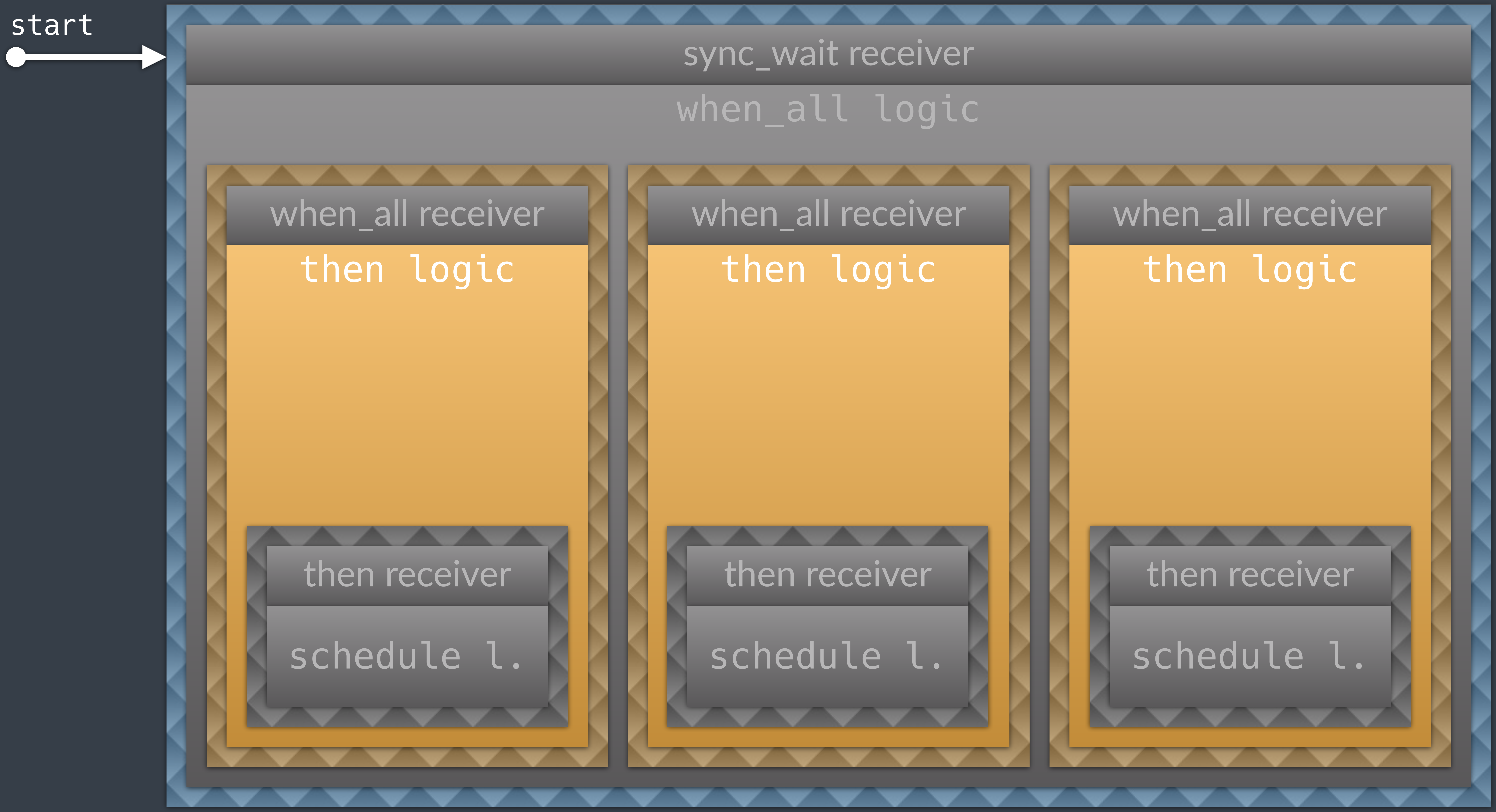
starting operation



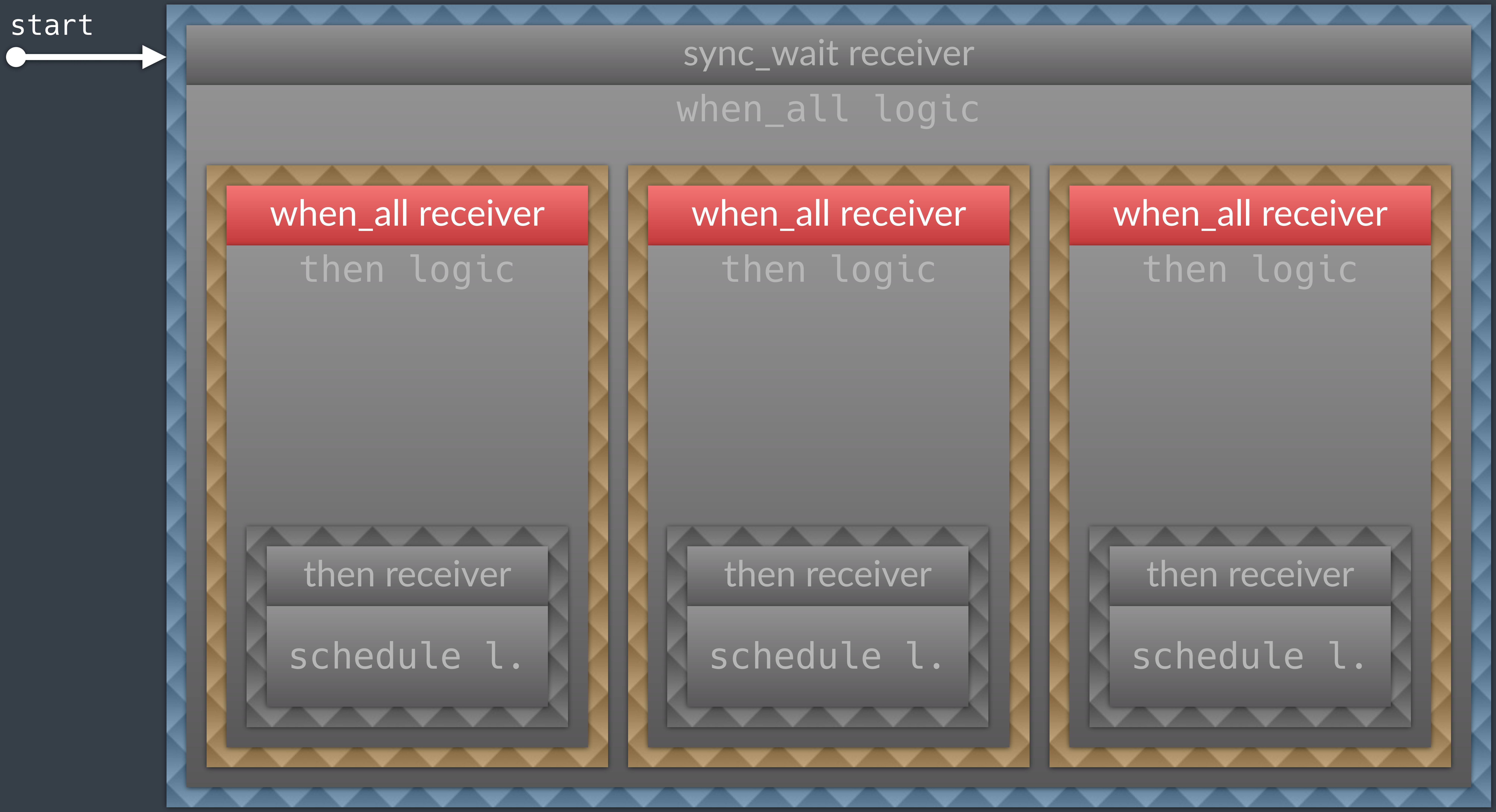
starting operation



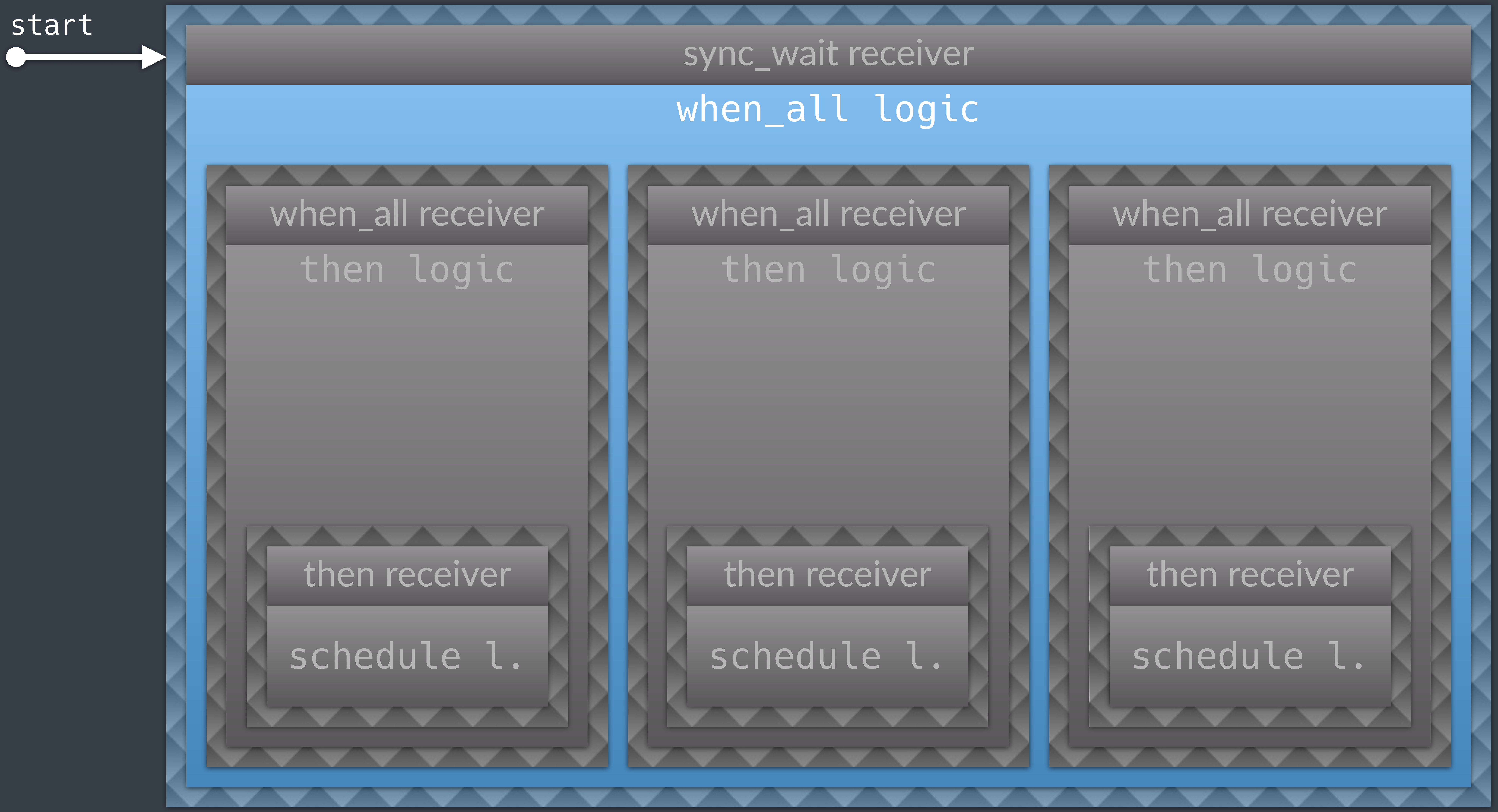
starting operation



starting operation

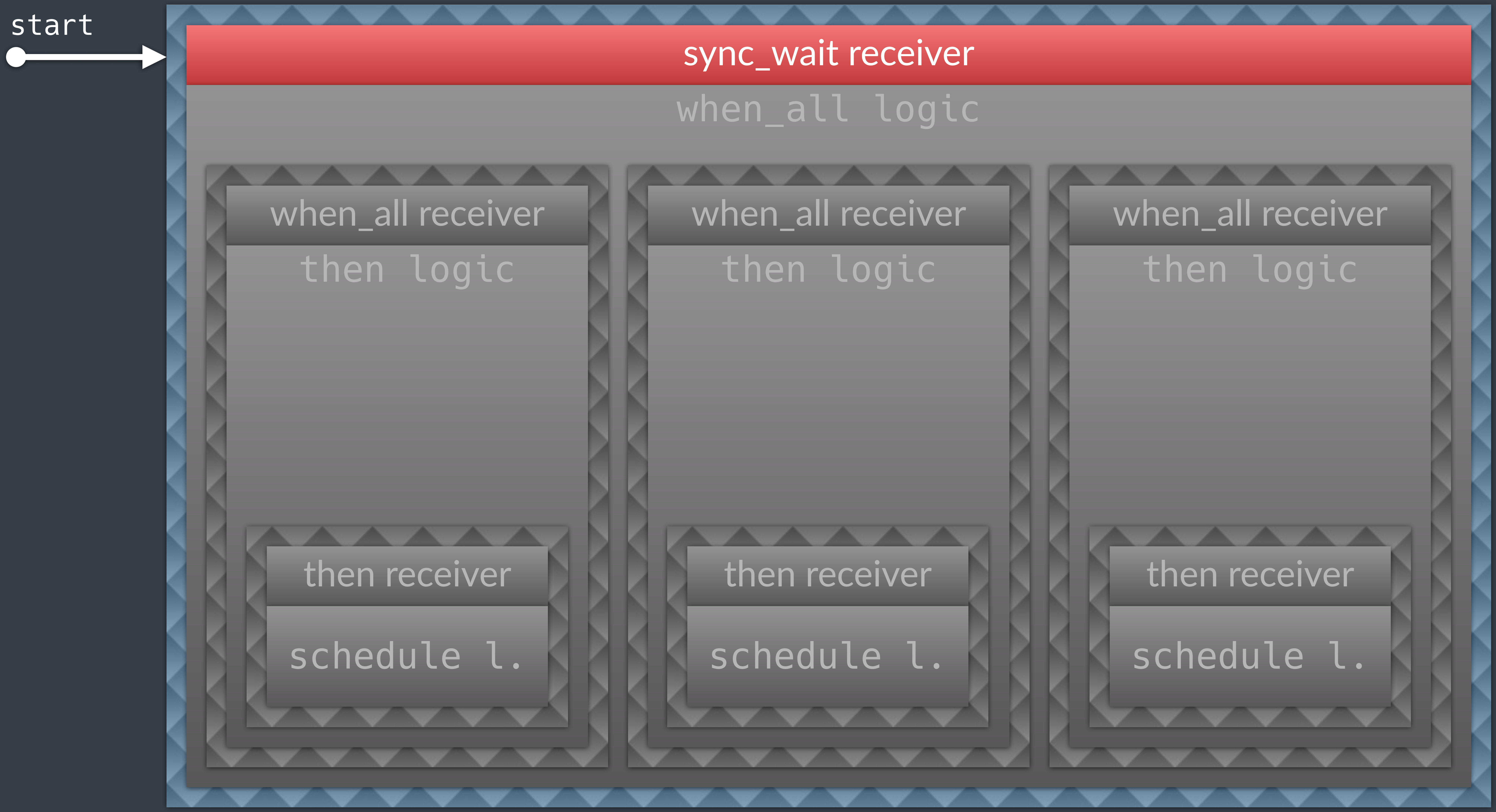


starting operation

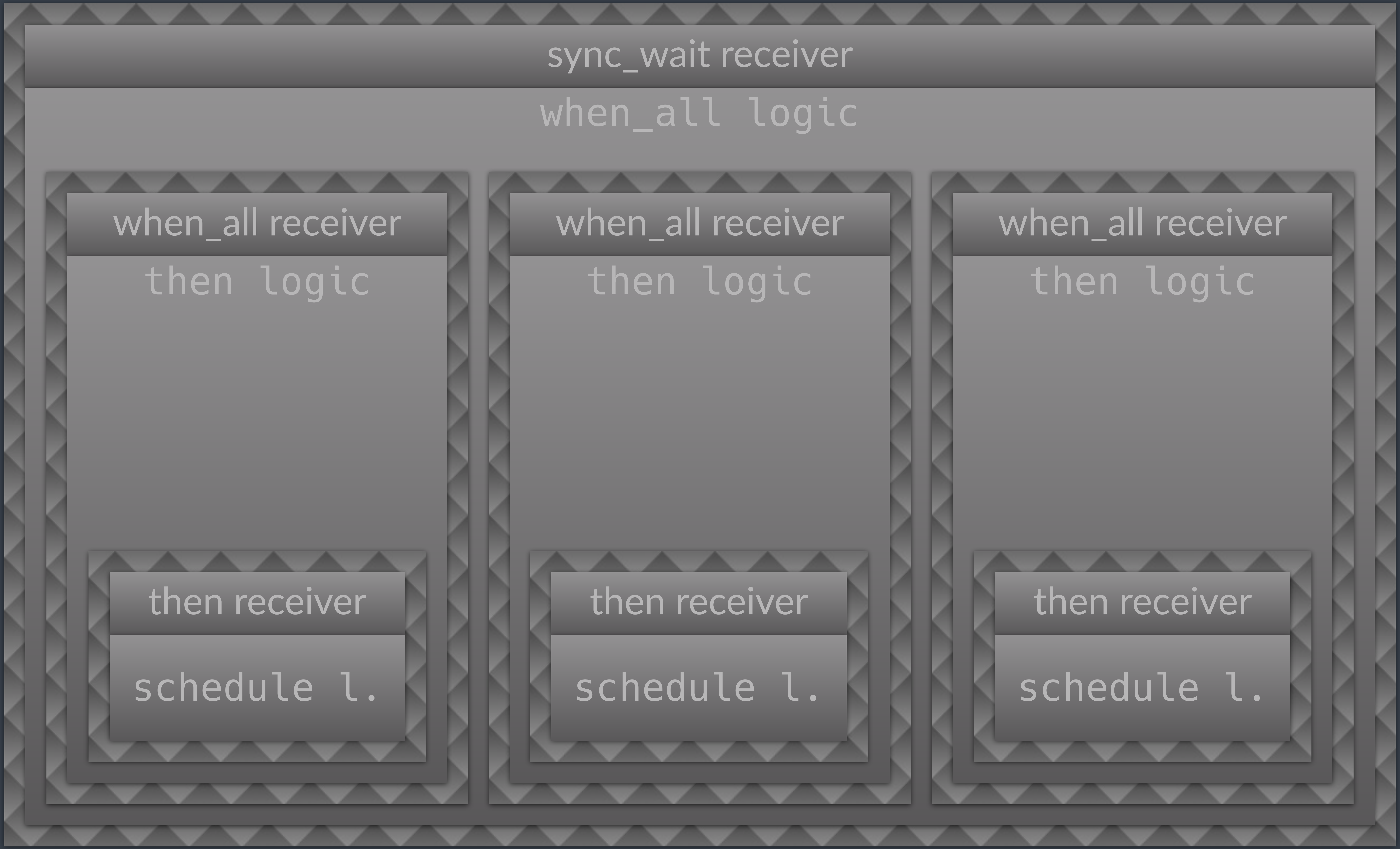


@LucT3o

starting operation



done


sync_wait receiver
when_all logic
when_all receiver
then logic
then receiver
schedule l.
when_all receiver
then logic
then receiver
schedule l.
when_all receiver
then logic
then receiver
schedule l.

focus of the talk

senders describe **computations**

functions	computations
same thread	entry thread ≠ exit thread

computations

generalisation of functions

computations are for **concurrency**
what functions are for **Structured** Programming

Structured Concurrency

5

functions

Structured Programming

computations

Structured Concurrency

functions	**senders**
Structured Programming	Structured Concurrency

some theoretical results

P2504R0

Computations as a global solution to concurrency

Published Proposal, 2021-12-11

Author:
Lucian Radu Teodorescu

Source:
GitHub

Issue Tracking:
GitHub

Project:
ISO/IEC JTC1/SC22/WG21 14882: Programming Language — C++

Audience:
SG1, LEWG

---

§ 1. Introduction

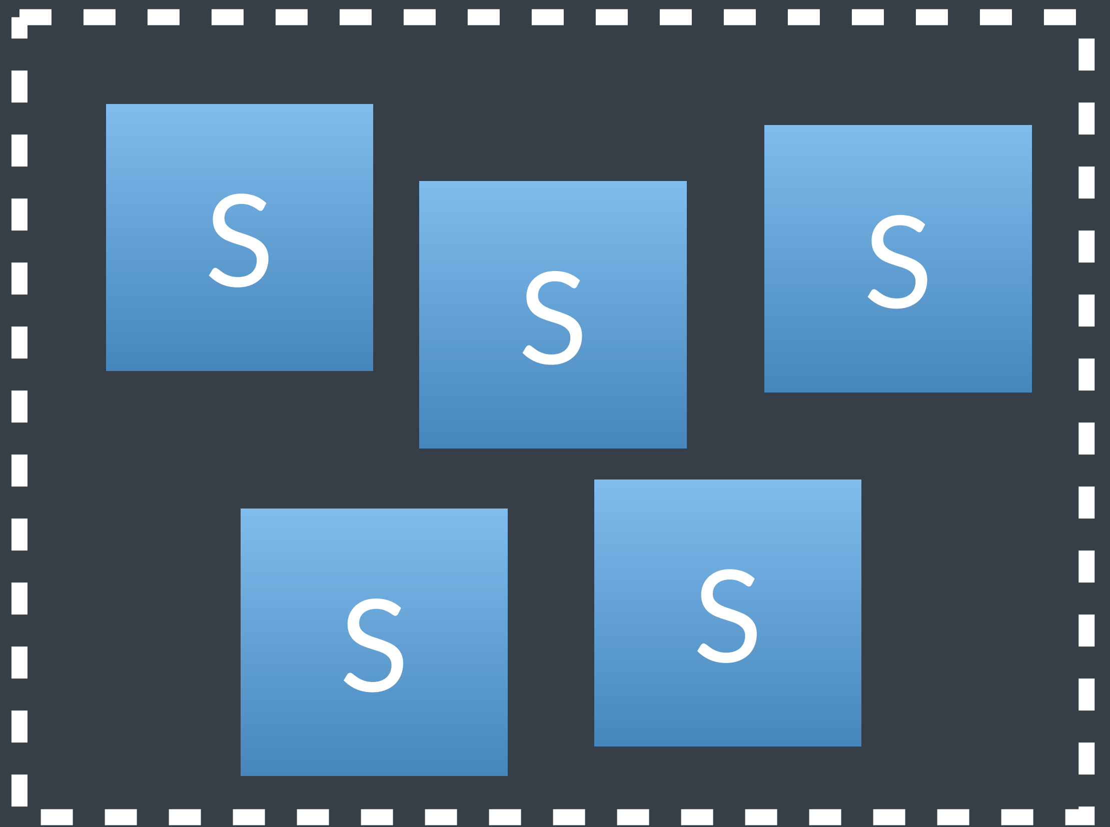
This paper aims at providing proof that the senders/receivers model proposed by [P2300R2] can constitute a global solution to concurrency.

http://wg21.link/P2504

some theoretical results (1)

all programs can be described in terms of senders

(w/o the need of synchronisation primitives)



some theoretical results (2)

any part of a program,

that has one entry point and one exit point,

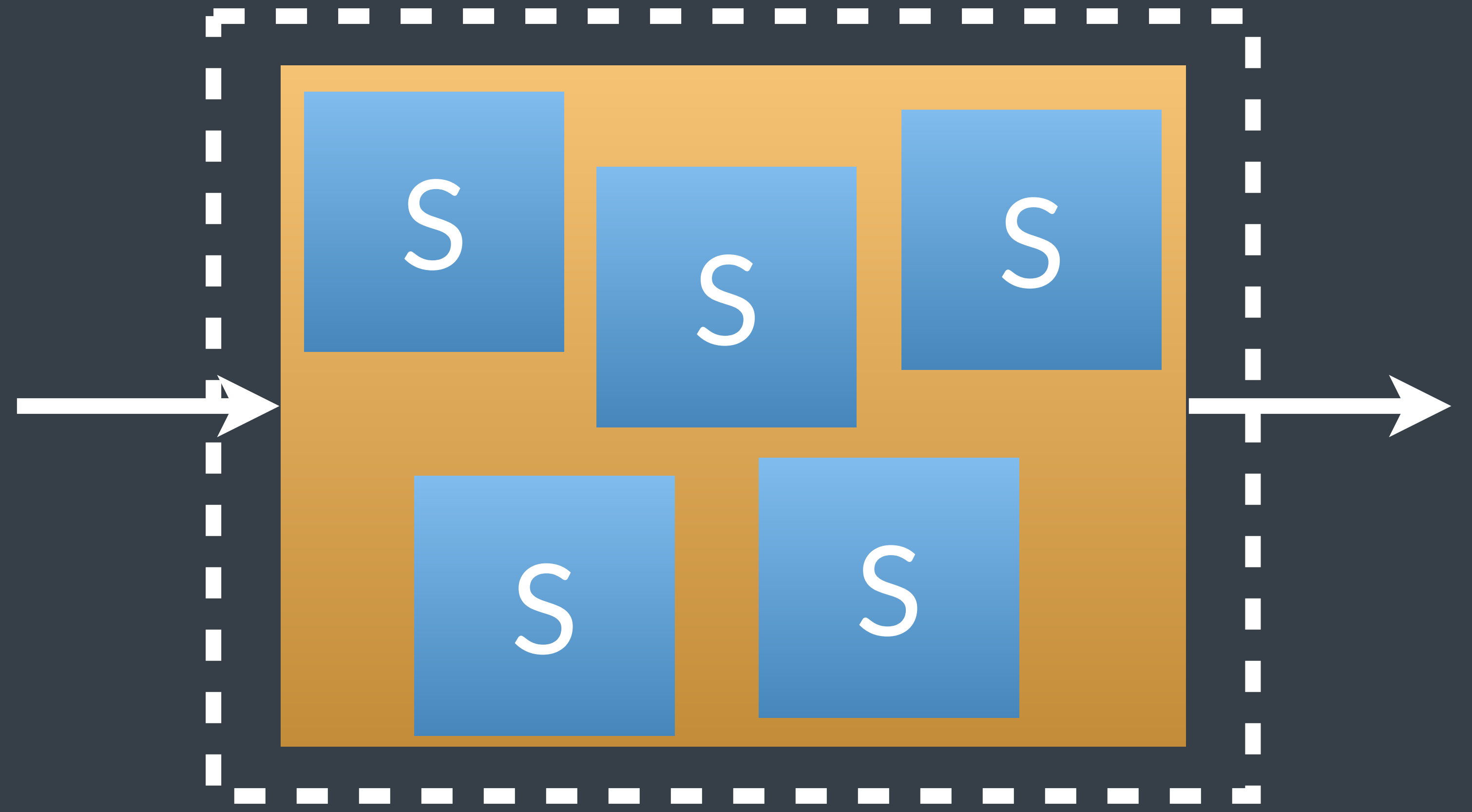
can be described as a sender

some theoretical results (3)

the entire program can be described as one sender

some theoretical results (4)

any sufficiently large concurrent chunk of work can be decomposed into smaller chunks of work, which can be described with senders



some theoretical results (5)

programs can be implemented using senders using maximum efficiency (under certain assumptions)

computations

fully encapsulate concurrency concerns

computations are for **concurrency**
what functions are for **Structured** Programming

Structured Concurrency

1. abstractions as building blocks
2. recursive decomposition
3. local reasoning
4. single entry, single exit point
5. soundness and completeness

1. abstractions as building blocks

use **computations** (senders)

similar to using functions

2. recursive decomposition

divide et impera

computations can be broken down into smaller computations

senders, from top to bottom

3. local reasoning

nested scopes

encapsulation of local concerns

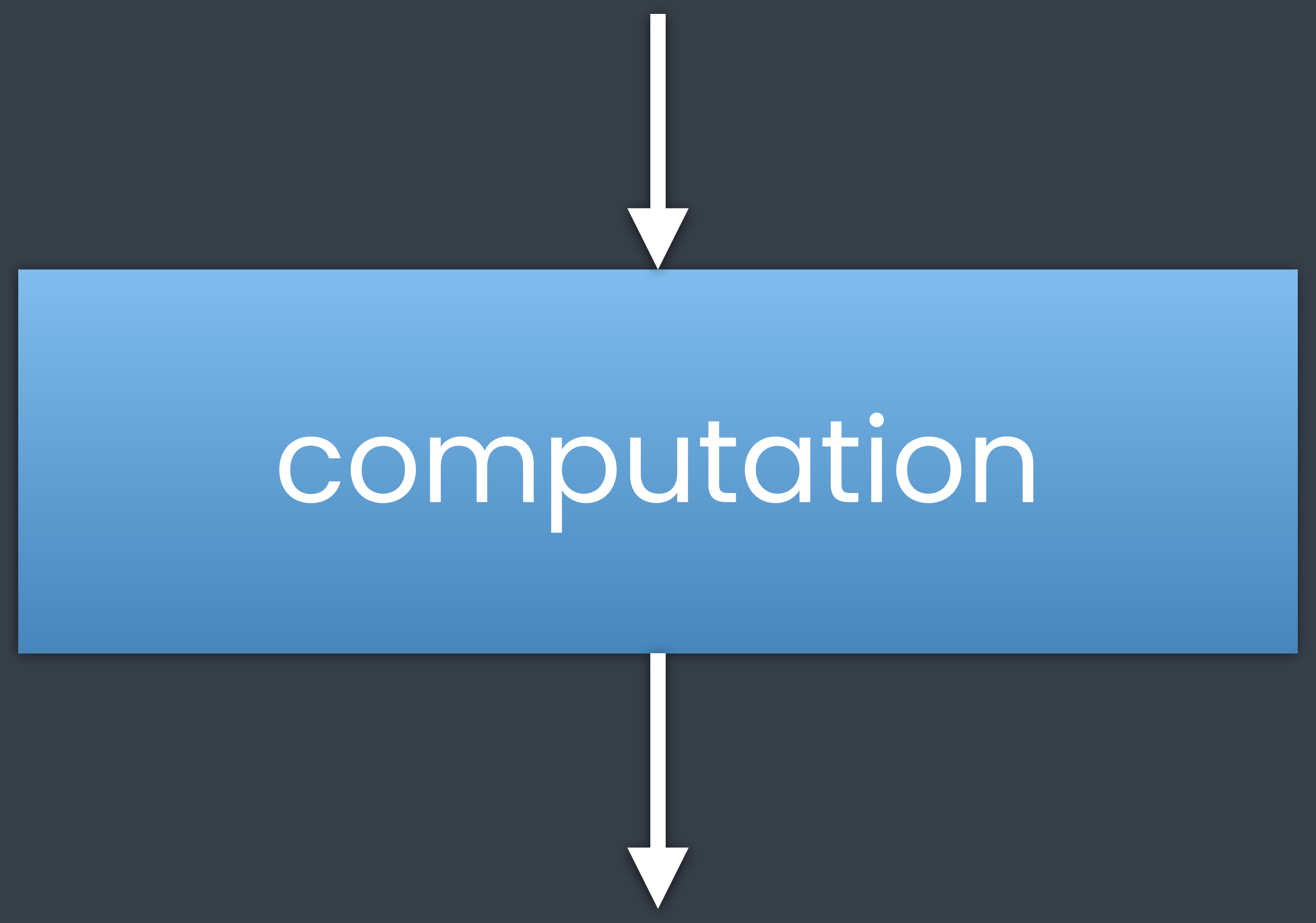
no spooky action at a distance

all the concerns are handled locally

inputs and outputs are clearly defined

senders nests

4. single entry, single exit point



by definition

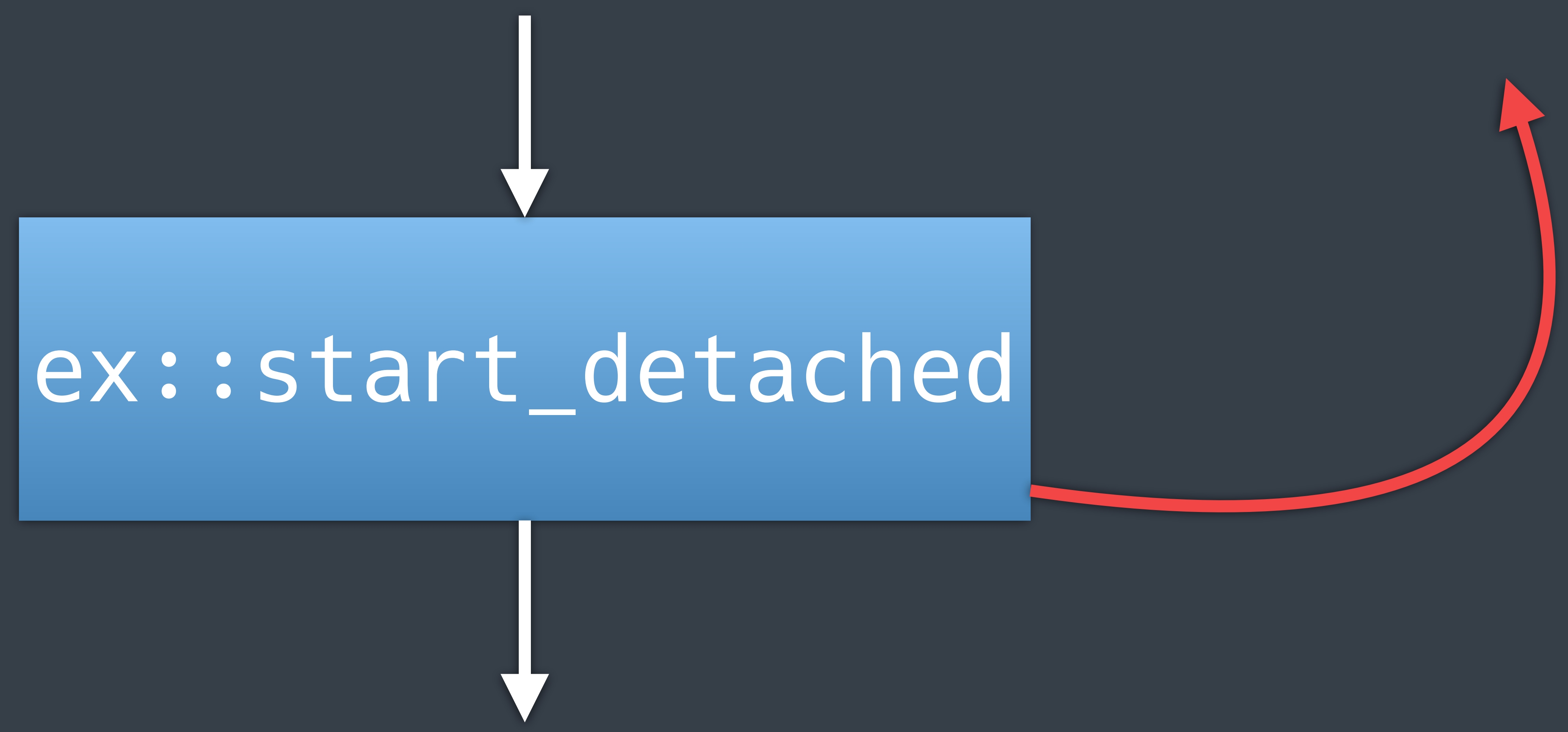
input:

```
- start(op_state)
```

output is one of:

```
- set_value(recv, values…)
- set_error(recv, error)
- set_stopped(recv)
```

discouraged: fire-and-forget



computations — same shape

composability of concurrent work

5. soundness and completeness

can Structured Concurrency be applied?

soundness

all programs can safely be built with senders
(without the need of sync. primitives)

completeness

all programs can be described by senders

Structured Concurrency

CHECK

1. abstractions as building blocks
2. recursive decomposition
3. local reasoning
4. single entry, single exit point
5. soundness and completeness

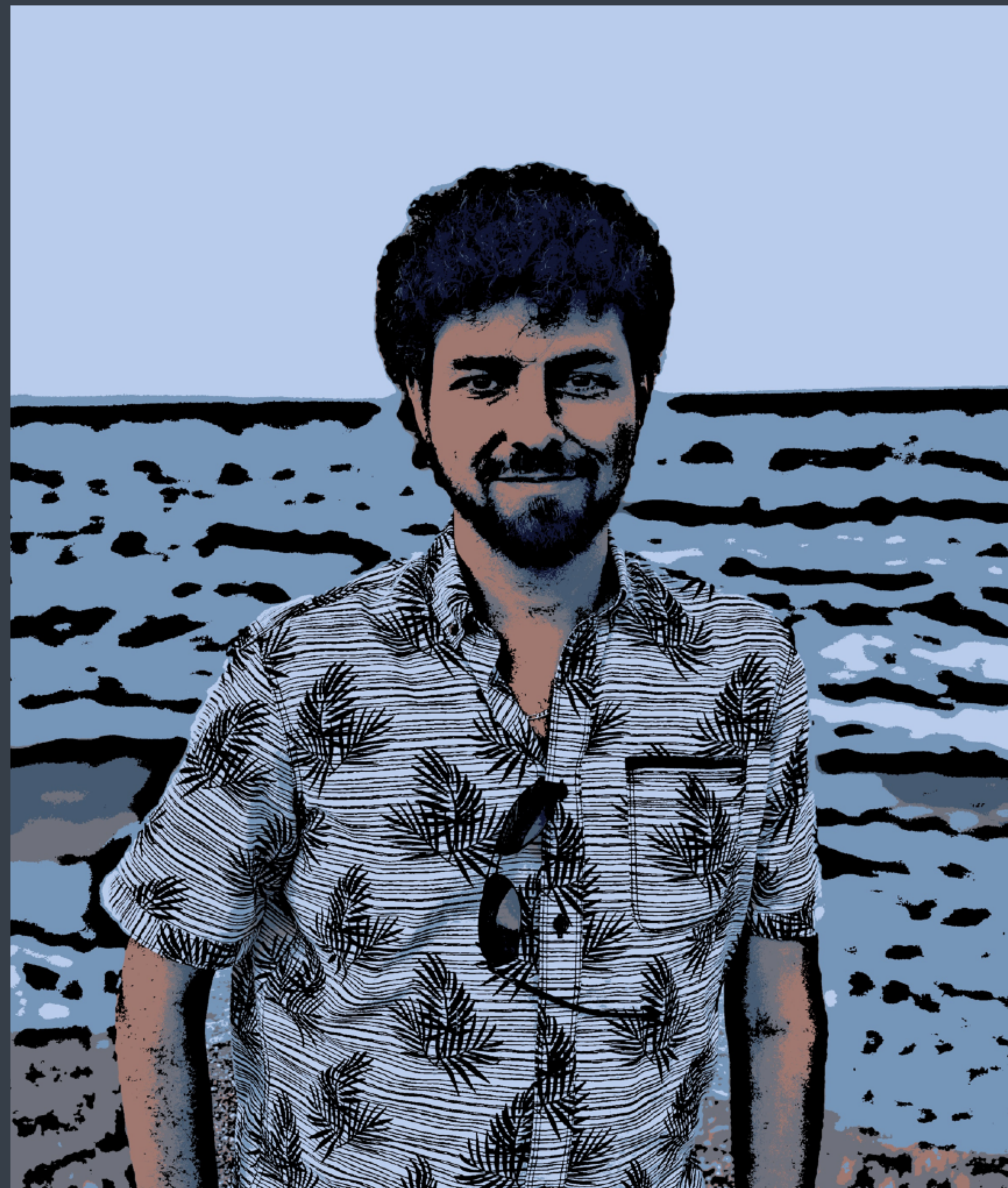
An Example

6

application

HTTP server

image processing



goals

senders as building blocks

recursive decomposition with senders

local reasoning

single entry and single exit point (*)

secondary goals

interaction with coroutines

type erasure for senders

https://github.com/lucteo/structured_concurrency_example

the entire app is a sender

```
auto main() -> int {
    auto [r] = std::this_thread::sync_wait(get_main_sender()).value();
    return r;
}
```

the entire app is a sender

```
auto get_main_sender() {
    return ex::just() | ex::then([] {
        //...
        return 0;
    });
}
```

top-level logic

```
auto get_main_sender() {
    return ex::just() | ex::then([] {
        int port = 8080;

        static_thread_pool pool{8};

        io::io_context ctx;
        set_sig_handler(ctx, SIGTERM);

        ex::sender auto snd = ex::on(ctx.get_scheduler(), listener(port, ctx, pool));
        ex::start_detached(std::move(snd));

        ctx.run();
        return 0;
    });
}
```

listener

```
auto listener(int port, io::io_context& ctx, static_thread_pool& pool)-> task<bool> {
    // ...
    co_return true;
}
```

listener

```
auto listener(int port, io::io_context& ctx, static_thread_pool& pool)-> task<bool> {
    io::listening_socket listen_sock;
    listen_sock.bind(port);
    listen_sock.listen();

    while (!ctx.is_stopped()) {
        io::connection conn = co_await io::async_accept(ctx, listen_sock);

        conn_data data{std::move(conn), ctx, pool};

        ex::sender auto snd =                                   //
                ex::just()                                      //
                | ex::let_value([data = std::move(data)]() {    //
                    return handle_connection(data);
                  });
        ex::start_detached(std::move(snd));
    }

    co_return true;
}
```

@LucT3o

async_accept -> a sender

```
inline auto async_accept(io_context& ctx, const listening_socket& sock)
        -> detail::async_accept_sender {
    return {&ctx, sock.fd()};
}
```

handle_connection

```
auto handle_connection(const conn_data& cdata) {
    return read_http_request(cdata.io_ctx_, cdata.conn_)
            | ex::transfer(cdata.pool_.get_scheduler())
            | ex::let_value([&cdata](http_server::http_request req) {
                return handle_request(cdata, std::move(req));
              })
            | ex::let_error([](std::exception_ptr) { return just_500_response(); })
            | ex::let_stopped([]() { return just_500_response(); })
            | ex::let_value([&cdata](http_server::http_response r) {
               return write_http_response(cdata.io_ctx_, cdata.conn_, std::move(r));
              });
}
```

just_500_response

```
auto just_500_response() {
    auto resp = http_server::create_response(
                    http_server::status_code::s_500_internal_server_error);
    return ex::just(std::move(resp));
}
```

read_http_request

```
auto read_http_request(io::io_context& ctx, const io::connection& conn)
        -> task<http_server::http_request> {
    http_server::request_parser parser;
    std::string buf;
    buf.reserve(1024 * 1024);
    io::out_buffer out_buf{buf};
    while (true) {
        std::size_t n = co_await io::async_read(ctx, conn, out_buf);
        auto data = std::string_view{buf.data(), n};
        auto r = parser.parse_next_packet(data);
        if (r)
            co_return {std::move(r.value())};
    }
}
```

async_read -> a sender

```
inline auto async_read(io_context& ctx, const connection& c, out_buffer buf)
        -> detail::async_read_sender {
    return {&ctx, c.fd(), buf};
}
```

write_http_request

```
auto write_http_response(io::io_context& ctx, const io::connection& conn,
        http_server::http_response resp) -> task<std::size_t> {
    std::vector<std::string_view> out_buffers;
    http_server::to_buffers(resp, out_buffers);
    std::size_t bytes_written{0};
    for (auto buf : out_buffers) {
        while (!buf.empty()) {
            auto n = co_await io::async_write(ctx, conn, buf);
            bytes_written += n;
            buf = buf.substr(n);
        }
    }
    co_return bytes_written;
}
```

async_write -> a sender

```
inline auto async_write(io_context& ctx, const connection& c, std::string_view data)
        -> detail::async_write_sender {
    return {&ctx, c.fd(), data};
}
```

handle_request

```
auto handle_request(const conn_data& cdata, http_server::http_request req)
        -> task<http_server::http_response> {
    auto puri = parse_uri(req.uri_);
    if (puri.path_ == "/transform/blur")
        co_return handle_blur(cdata, std::move(req), puri);
    else if (puri.path_ == "/transform/adaptthresh")
        co_return handle_adaptthresh(cdata, std::move(req), puri);
    else if (puri.path_ == "/transform/reducecolors")
        co_return handle_reducecolors(cdata, std::move(req), puri);
    else if (puri.path_ == "/transform/cartoonify")
        co_return co_await handle_cartoonify(cdata, std::move(req), puri);
    else if (puri.path_ == "/transform/oilpainting")
        co_return handle_oilpainting(cdata, std::move(req), puri);
    else if (puri.path_ == "/transform/contourpaint")
        co_return co_await handle_contourpaint(cdata, std::move(req), puri);
   co_return http_server::create_response(http_server::status_code::s_404_not_found);
}
```

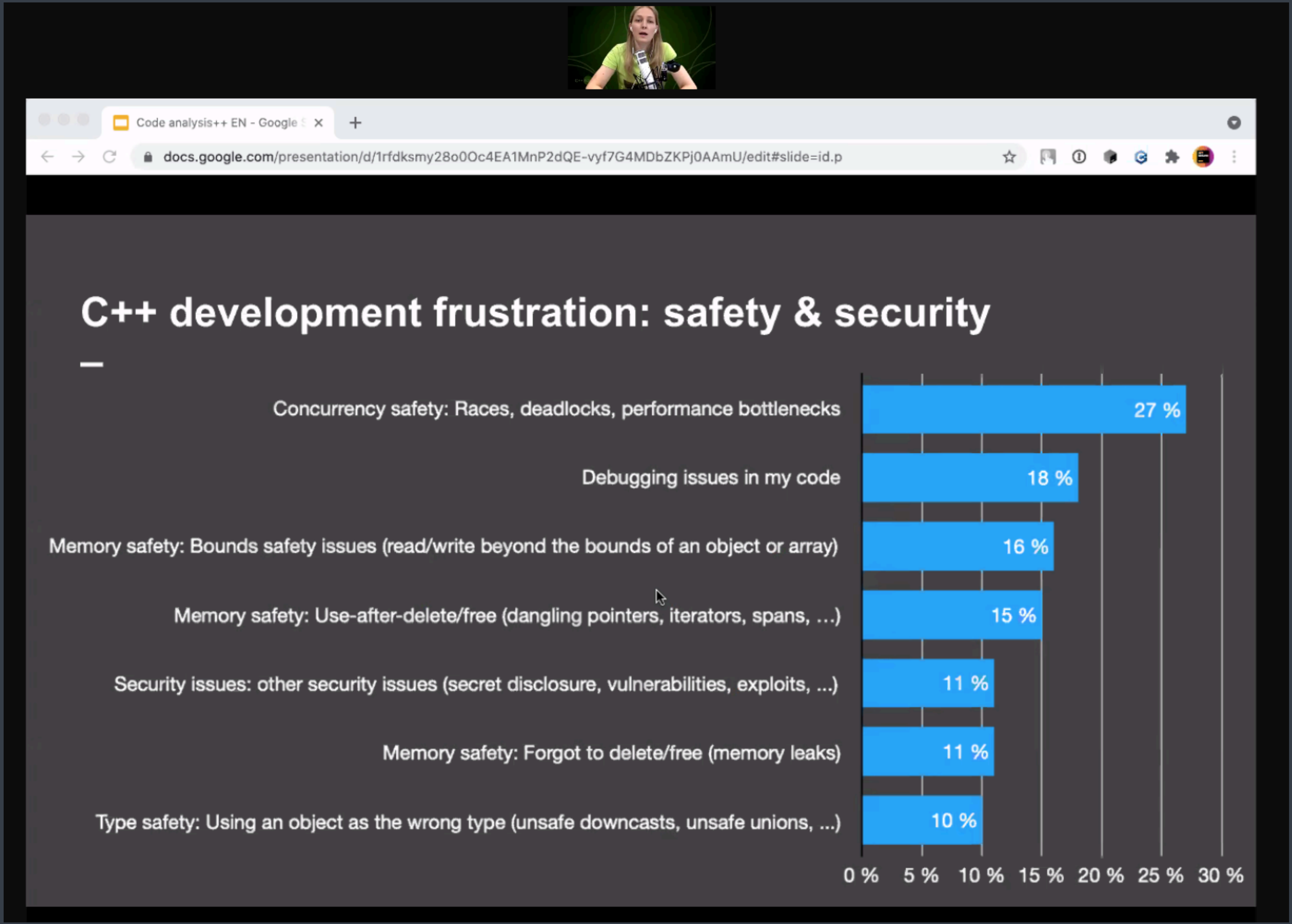
```
auto handle_cartoonify(const conn_data& cdata, http_server::http_request&& req, parsed_uri puri)
        -> task<http_server::http_response> {
    int blur_size = get_param_int(puri, "blur_size", 3);
    int num_colors = get_param_int(puri, "num_colors", 5);
    int block_size = get_param_int(puri, "block_size", 5);
    int diff = get_param_int(puri, "diff", 5);
    auto src = to_cv(req.body_);
    ex::sender auto snd = ex::when_all(
        ex::transfer_just(cdata.pool_.get_scheduler(), src)
            | ex::then([=](const cv::Mat& src) {
                auto gray = tr_to_grayscale(tr_blur(src, blur_size));
                return tr_adaptthresh(gray, block_size, diff);
            }),
        ex::transfer_just(cdata.pool_.get_scheduler(), src)
                | ex::then([=](const cv::Mat& src) {
                    return tr_reducecolors(src, num_colors);
                })
        )
      | ex::then([](const cv::Mat& edges, const cv::Mat& reduced_colors) {
            return tr_apply_mask(reduced_colors, edges);
        })
      | ex::then(img_to_response);
    co_return co_await std::move(snd);
}
```

recursive decomposition

- get_main_sender
  - listener
    - async_accept
    - handle_connection
      - read_http_request
        - async_read
      - handle_request
        - handle_cartoonify
        - …
      - just_500_response
      - write_http_response
        - async_write

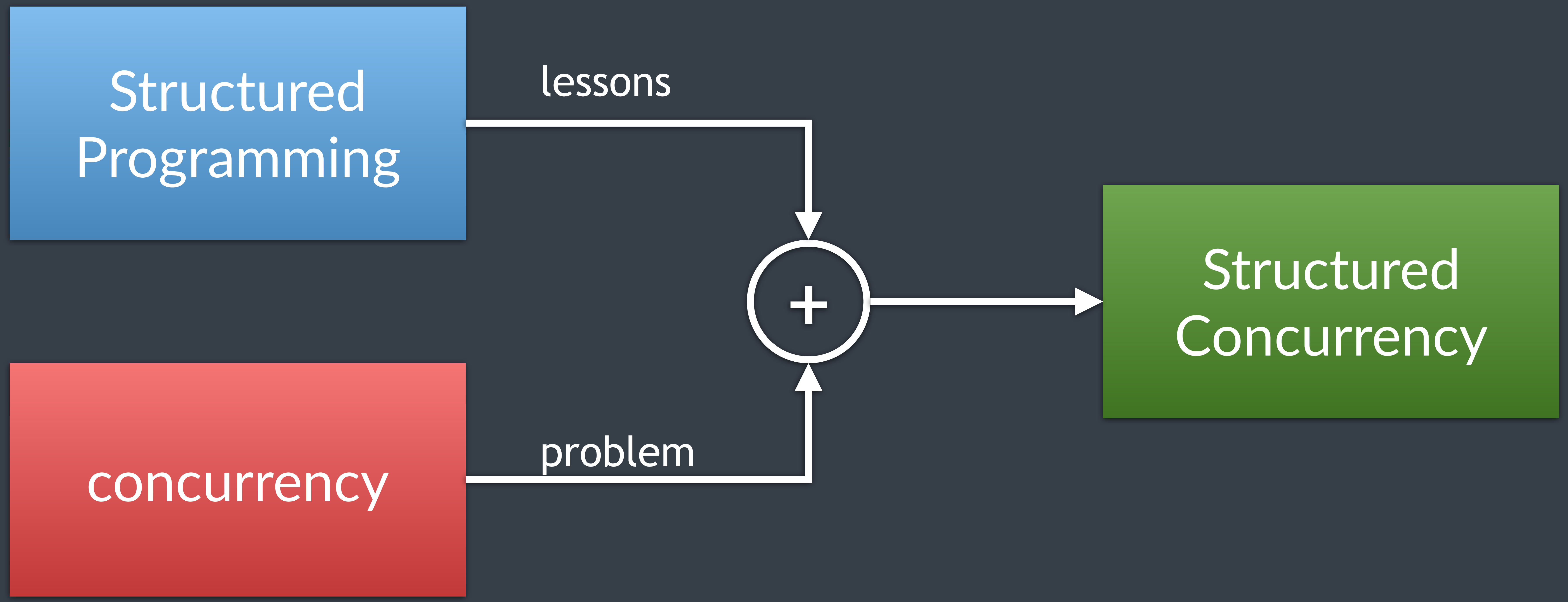
7

Conclusions


Code analysis++ EN - Google S
docs.google.com/presentation/d/1rfdksmy28o0Oc4EA1MnP2dQE-vyf7G4MDbZKPj0AAmU/edit#slide=id.p
C++ development frustration: safety & security
Concurrency safety: Races, deadlocks, performance bottlenecks
27 %
Debugging issues in my code
18 %
Memory safety: Bounds safety issues (read/write beyond the bounds of an object or array)
16 %
Memory safety: Use-after-delete/free (dangling pointers, iterators, spans, ...)
15 %
Security issues: other security issues (secret disclosure, vulnerabilities, exploits, ...)
11 %
Memory safety: Forgot to delete/free (memory leaks)
11 %
Type safety: Using an object as the wrong type (unsafe downcasts, unsafe unions, ...)
10 %
0 %
5 %
10 %
15 %
20 %
25 %
30 %

@LucT3o

what we did



Structured Programming
lessons
concurrency
problem
+
Structured Concurrency

key insight

functions	→	Structured Programming
computations	→	Structured Concurrency

Structured Concurrency

1. abstractions as building blocks
2. recursive decomposition
3. local reasoning
4. single entry, single exit point
5. soundness and completeness


Thank You
@LucT3o
lucteo.ro