srsRAN Documentation

Sched.cc

Method Name:

void sched::init()

Parameters:

rrc_interface_mac* rrc_

const sched_args_t& sched_cfg_

Functionality:

Initializes the radio resource control

- This is a protocol used between UE and base stations, defines states that a UE can be in.

Initializes scheduler arguments and configuration

Brings in parameters rrc and sched cfg for use

Initializes first carrier scheduler

Returns:

void

Code:

```
void sched::init(rrc_interface_mac* rrc_, const sched_args_t& sched_cfg_)
{
  rrc       = rrc_;
  sched_cfg = sched_cfg_;

  // Initialize first carrier scheduler
  carrier_schedulers.emplace_back(new carrier_sched{rrc, &ue_db, 0, &sched_results});

  reset();
}
```

Method Name:

int sched::reset()

Parameters:

none

Functionality:

- Resets the scheduler and clears current UEs

Returns:

0

Code:

```
int sched::reset()
{
  std::lock_guard<std::mutex> lock(sched_mutex);
  for (std::unique_ptr<carrier_sched>& c : carrier_schedulers) {
    c->reset();
  }
  ue_db.clear();
  return 0;
}
```

Method Name:

int sched::cell_cfg()

Parameters:

const std::vector<sched_interface::cell_cfg_t>& cell_cfg

Functionality:

Resizes cell size to meet carrier requirements

Returns error is index of cell is out of range

Sets number of allowed carriers by the cell size for the scheduler

Sets up schedulers configuration based on cell size and allowed carriers

Returns:

SRSRAN_ERROR

0

Code:

```
/// Called by rrc::init
int sched::cell_cfg(const std::vector<sched_interface::cell_cfg_t>& cell_cfg)
{
  std::lock_guard<std::mutex> lock(sched_mutex);
  // Setup derived config params
  sched_cell_params.resize(cell_cfg.size());
  for (uint32_t cc_idx = 0; cc_idx < cell_cfg.size(); ++cc_idx) {
    if (not sched_cell_params[cc_idx].set_cfg(cc_idx, cell_cfg[cc_idx], sched_cfg)) {
      return SRSRAN_ERROR;
    }
  }

  sched_results.set_nof_carriers(cell_cfg.size());

  // Create remaining cells, if not created yet
  uint32_t prev_size = carrier_schedulers.size();
  carrier_schedulers.resize(sched_cell_params.size());
  for (uint32_t i = prev_size; i < sched_cell_params.size(); ++i) {
    carrier_schedulers[i].reset(new carrier_sched{rrc, &ue_db, i, &sched_results});
  }

  // setup all carriers cfg params
  for (uint32_t i = 0; i < sched_cell_params.size(); ++i) {
    carrier_schedulers[i]->carrier_cfg(sched_cell_params[i]);
  }

  configured = true;
  return 0;
}
```

Method Name:

int sched::ue_cfg()

Parameters:

uint16_t rnti

const sched_interface::ue_cfg_t& ue_cfg

Functionality:

Configures existing UE

Allows for configuration of new UE

Returns:

SRSRAN_SUCCESS

Code:

```
int sched::ue_cfg(uint16_t rnti, const sched_interface::ue_cfg_t& ue_cfg)
{
  {
    // config existing user
    std::lock_guard<std::mutex> lock(sched_mutex);
    auto                        it = ue_db.find(rnti);
    if (it != ue_db.end()) {
      it->second->set_cfg(ue_cfg);
      return SRSRAN_SUCCESS;
    }
  }

  // Add new user case
  std::unique_ptr<sched_ue>   ue{new sched_ue(rnti, sched_cell_params, ue_cfg)};
  std::lock_guard<std::mutex> lock(sched_mutex);
  ue_db.insert(std::make_pair(rnti, std::move(ue)));
  return SRSRAN_SUCCESS;
}
```

Method Name:

int sched::ue_rem()

Parameters:

uint16_t rnti

Functionality:

Removes UE from connection

Checks to see if UE count is over 0 and then removes that UE

Uses rnti to identify which UE to check and remove

Returns:

SRSRAN_ERROR

SRSRAN_SUCCESS

Code:

```
int sched::ue_rem(uint16_t rnti)
{
  std::lock_guard<std::mutex> lock(sched_mutex);
  if (ue_db.count(rnti) > 0) {
    ue_db.erase(rnti);
  } else {
    Error("User rnti=0x%x not found", rnti);
    return SRSRAN_ERROR;
  }
  return SRSRAN_SUCCESS;
}
```

Method Name:

bool sched::ue_exists()

Parameters:

uint16_t rnti

Functionality:

Checks if any UE are connected

Returns:

True

False

Code:

```
bool sched::ue_exists(uint16_t rnti)
{
  return ue_db_access_locked(
              rnti, [](sched_ue& ue) {}, nullptr, false) >= 0;
}
```

Method Name:

void sched::phy_config_enabled()

Parameters:

uint16_t rnti

bool enabled

Functionality:

Enables physical layer connection

Returns:

void

Code:

```
void sched::phy_config_enabled(uint16_t rnti, bool enabled)
{
  // TODO: Check if correct use of last_tti
  ue_db_access_locked(
      rnti, [this, enabled](sched_ue& ue) { ue.phy_config_enabled(last_tti, enabled); }, __PRETTY_FUNCTION__);
}
```

Method Name:

int sched::bearer_ue_cfg()

Parameters:

uint16_t rnti

uint32_t lc_id

const sched_interface::ue_bearer_cfg_t& cfg_

Functionality:

Configures the UE for bearer splitting of the data

Returns:

Configuration for bearer split

Code:

```
int sched::bearer_ue_cfg(uint16_t rnti, uint32_t lc_id, const sched_interface::ue_bearer_cfg_t& cfg_)
{
  return ue_db_access_locked(rnti, [lc_id, cfg_](sched_ue& ue) { ue.set_bearer_cfg(lc_id, cfg_); });
}
```

Method Name:

int sched::bearer_ue_rem()

Parameters:

uint16_t rnti

uint32_t lc_id

Functionality:

Removes configuration for bearer splitting

Returns:

Int for removing bearer splitting

Code:

```
int sched::bearer_ue_rem(uint16_t rnti, uint32_t lc_id)
{
  return ue_db_access_locked(rnti, [lc_id](sched_ue& ue) { ue.rem_bearer(lc_id); });
}
```

Method Name:

uint32_t sched::get_dl_buffer()

Parameters:

uint16_t rnti

Functionality:

Sets ret to adjust for RLC download data

Returns:

ret

Code:

```
uint32_t sched::get_dl_buffer(uint16_t rnti)
{
  uint32_t ret = SRSRAN_ERROR;
  ue_db_access_locked(
      rnti, [&ret](sched_ue& ue) { ret = ue.get_pending_dl_rlc_data(); }, __PRETTY_FUNCTION__);
  return ret;
}
```

Method Name:

uint32_t sched::get_ul_buffer()

Parameters:

uint16_t rnti

Functionality:

Sets ret to adjust for RLC upload data

Returns:

ret

Code:

```
uint32_t sched::get_ul_buffer(uint16_t rnti)
{
  // TODO: Check if correct use of last_tti
  uint32_t ret = SRSRAN_ERROR;
  ue_db_access_locked(
      rnti,
      [this, &ret](sched_ue& ue) { ret = ue.get_pending_ul_new_data(to_tx_ul(last_tti), -1); },
      __PRETTY_FUNCTION__);
  return ret;
}
```

Method Name:

int sched::dl_rlc_buffer_state()

Parameters:

uint16_t rnti

uint32_t lc_id

uint32_t tx_queue

uint32_t retx_queue

Functionality:

Gets current buffer state for RLC

Returns:

Download buffer state

Code:

```
int sched::dl_rlc_buffer_state(uint16_t rnti, uint32_t lc_id, uint32_t tx_queue, uint32_t retx_queue)
{
  return ue_db_access_locked(rnti, [&](sched_ue& ue) { ue.dl_buffer_state(lc_id, tx_queue, retx_queue); });
}
```

Method Name:

int sched::dl_mac_buffer_state()

Parameters:

uint16_t rnti

uint32_t ce_code

uint32_t nof_cmds

Functionality:

Gets current buffer state for MAC layer

Returns:

MAC buffer state

Code:

```
int sched::dl_mac_buffer_state(uint16_t rnti, uint32_t ce_code, uint32_t nof_cmds)
{
  return ue_db_access_locked(rnti, [ce_code, nof_cmds](sched_ue& ue) { ue.mac_buffer_state(ce_code, nof_cmds); });
}
```

Method Name:

int sched::dl_ack_info()

Parameters:

uint32_t tti_rx

uint16_t rnti

uint32_t enb_cc_idx

uint32_t tb_idx

bool ack

Functionality:

Sets ret to -1

Reassigns ret based on acknowledgement of UE

Returns:

ret

Code:

```
int sched::dl_ack_info(uint32_t tti_rx, uint16_t rnti, uint32_t enb_cc_idx, uint32_t tb_idx, bool ack)
{
  int ret = -1;
  ue_db_access_locked(
      rnti,
      [&](sched_ue& ue) { ret = ue.set_ack_info(tti_point{tti_rx}, enb_cc_idx, tb_idx, ack); },
      __PRETTY_FUNCTION__);
  return ret;
}
```

Method Name:

int sched::ul_crc_info()

Parameters:

uint32_t tti_rx

uint16_t rnti

uint32_t enb_cc_idx

bool crc

Functionality:

Gives sched class the UE upload crc information

Returns:

Upload crc information

Code:

```
int sched::ul_crc_info(uint32_t tti_rx, uint16_t rnti, uint32_t enb_cc_idx, bool crc)
{
  return ue_db_access_locked(
      rnti, [tti_rx, enb_cc_idx, crc](sched_ue& ue) { ue.set_ul_crc(tti_point{tti_rx}, enb_cc_idx, crc); });
}
```

Method Name:

int sched::dl_ri_info()

Parameters:

uint32_t tti

uint16_t rnti

uint32_t enb_cc_idx

uint32_t ri_value

Functionality:

Sets download radio information based on tti point

Returns:

Radio download information

Code:

```
int sched::dl_ri_info(uint32_t tti, uint16_t rnti, uint32_t enb_cc_idx, uint32_t ri_value)
{
  return ue_db_access_locked(
      rnti, [tti, enb_cc_idx, ri_value](sched_ue& ue) { ue.set_dl_ri(tti_point{tti}, enb_cc_idx, ri_value); });
}
```

Method Name:

int sched::dl_pmi_info()

Parameters:

uint32_t tti

uint16_t rnti

uint32_t enb_cc_idx

uint32_t pmi_value

Functionality:

Gets UE download pmi information

Returns:

Download pmi information

Code:

```
int sched::dl_pmi_info(uint32_t tti, uint16_t rnti, uint32_t enb_cc_idx, uint32_t pmi_value)
{
  return ue_db_access_locked(
      rnti, [tti, enb_cc_idx, pmi_value](sched_ue& ue) { ue.set_dl_pmi(tti_point{tti}, enb_cc_idx, pmi_value); });
}
```

Method Name:

int sched::dl_cqi_info()

Parameters:

uint32_t tti

uint16_t rnti

uint32_t enb_cc_idx

uint32_t cqi_value

Functionality:

Gets download cqi information

Returns:

Download cqi information

Code:

```
int sched::dl_cqi_info(uint32_t tti, uint16_t rnti, uint32_t enb_cc_idx, uint32_t cqi_value)
{
  return ue_db_access_locked(
      rnti, [tti, enb_cc_idx, cqi_value](sched_ue& ue) { ue.set_dl_cqi(tti_point{tti}, enb_cc_idx, cqi_value); });
}
```

Method Name:

int sched::dl_rach_info()

Parameters:

uint32_t enb_cc_idx

dl_sched_rar_info_t rar_info

Functionality:

Gets RACH info for downloading

Returns:

Download RACH information

Code:

```
int sched::dl_rach_info(uint32_t enb_cc_idx, dl_sched_rar_info_t rar_info)
{
  std::lock_guard<std::mutex> lock(sched_mutex);
  return carrier_schedulers[enb_cc_idx]->dl_rach_info(rar_info);
}
```

Method Name:

int sched::ul_snr_info()

Parameters:

uint32_t tti_rx

uint16_t rnti

uint32_t enb_cc_idx

float snr

uint32_t ul_ch_code

Functionality:

Gets SNR information

Returns:

SNR information

Code:

```
int sched::ul_snr_info(uint32_t tti_rx, uint16_t rnti, uint32_t enb_cc_idx, float snr, uint32_t ul_ch_code)
{
  return ue_db_access_locked(rnti,
                             [&](sched_ue& ue) { ue.set_ul_snr(tti_point{tti_rx}, enb_cc_idx, snr, ul_ch_code); });
}
```

Method Name:

int sched::ul_bsr()

Parameters:

uint16_t rnti

uint32_t lcg_id

uint32_t bsr

Functionality:

Gets buffer status for upload

Returns:

Upload buffer status

Code:

```
int sched::ul_bsr(uint16_t rnti, uint32_t lcg_id, uint32_t bsr)
{
  return ue_db_access_locked(rnti, [lcg_id, bsr](sched_ue& ue) { ue.ul_buffer_state(lcg_id, bsr); });
}
```

Method Name:

int sched::ul_buffer_add()

Parameters:

uint16_t rnti

uint32_t lcid

uint32_t bytes

Functionality:

Adds buffer to UE upload

Returns:

Bytes of buffer added

Code:

```
int sched::ul_buffer_add(uint16_t rnti, uint32_t lcid, uint32_t bytes)
{
  return ue_db_access_locked(rnti, [lcid, bytes](sched_ue& ue) { ue.ul_buffer_add(lcid, bytes); });
}
```

Method Name:

int sched::ul_phr()

Parameters:

uint16_t rnti

int phr

Functionality:

Get PHR information

Returns:

PHR information

Code:

```
int sched::ul_phr(uint16_t rnti, int phr)
{
  return ue_db_access_locked(
      rnti, [phr](sched_ue& ue) { ue.ul_phr(phr); }, __PRETTY_FUNCTION__);
}
```

Method Name:

int sched::ul_sr_info()

Parameters:

uint32_t tti

uint16_t rnti

Functionality:

Sets segment routing for upload

Returns:

Segment route information

Code:

```
int sched::ul_sr_info(uint32_t tti, uint16_t rnti)
{
  return ue_db_access_locked(
      rnti, [](sched_ue& ue) { ue.set_sr(); }, __PRETTY_FUNCTION__);
}
```

Method Name:

void sched::set_dl_tti_mask()

Parameters:

uint8_t* tti_mask

uint32_t nof_sfs

Functionality:

Sets tti mask for download on first scheduler

Returns:

void

Code:

```
void sched::set_dl_tti_mask(uint8_t* tti_mask, uint32_t nof_sfs)
{
  std::lock_guard<std::mutex> lock(sched_mutex);
  carrier_schedulers[0]->set_dl_tti_mask(tti_mask, nof_sfs);
}
```

Method Name:

std::array<int, SRSRAN_MAX_CARRIERS> sched::get_enb_ue_cc_map()

Parameters:

uint16_t rnti

Functionality:

Set ret to -1 for inactive carriers

Fills cells with UEs

Adjusts ret to meet UE needs

Returns:

ret

Code:

```
std::array<int, SRSRAN_MAX_CARRIERS> sched::get_enb_ue_cc_map(uint16_t rnti)
{
  std::array<int, SRSRAN_MAX_CARRIERS> ret{};
  ret.fill(-1); // -1 for inactive & non-existent carriers
  ue_db_access_locked(
      rnti,
      [this, &ret](sched_ue& ue) {
        for (size_t enb_cc_idx = 0; enb_cc_idx < carrier_schedulers.size(); ++enb_cc_idx) {
          const sched_ue_cell* cc_ue = ue.find_ue_carrier(enb_cc_idx);
          if (cc_ue != nullptr) {
            ret[enb_cc_idx] = cc_ue->get_ue_cc_idx();
          }
        }
      },
      __PRETTY_FUNCTION__);
  return ret;
}
```

Method Name:

std::array<bool, SRSRAN_MAX_CARRIERS> sched::get_scell_activation_mask()

Parameters:

uint16_t rnti

Functionality:

Finds UE carriers

Adds scheduler cell mask to each UE carrier

Returns:

Scheduler cell masks

Code:

```
std::array<bool, SRSRAN_MAX_CARRIERS> sched::get_scell_activation_mask(uint16_t rnti)
{
  std::array<bool, SRSRAN_MAX_CARRIERS> scell_mask = {};
  ue_db_access_locked(rnti, [this, &scell_mask](sched_ue& ue) {
    for (size_t enb_cc_idx = 0; enb_cc_idx < carrier_schedulers.size(); ++enb_cc_idx) {
      const sched_ue_cell* cc_ue = ue.find_ue_carrier(enb_cc_idx);
      if (cc_ue != nullptr and (cc_ue->cc_state() == cc_st::active or cc_ue->cc_state() == cc_st::activating)) {
        scell_mask[cc_ue->get_ue_cc_idx()] = true;
      }
    }
  });
  return scell_mask;
}
```

Method Name:

int sched::dl_sched()

Parameters:

uint32_t tti_tx_dl

uint32_t enb_cc_idx

sched_interface::dl_sched_res_t& sched_result

Functionality:

Checks download for configuration

Checks for base station carrier indexing outside of schedulers carrier size

Sets tti receive to tti transmit minus base station transmit delay

Sets new tti to tti receive

Get scheduler result

Returns:

0

Code:

```
// Downlink Scheduler API
int sched::dl_sched(uint32_t tti_tx_dl, uint32_t enb_cc_idx, sched_interface::dl_sched_res_t& sched_result)
{
  std::lock_guard<std::mutex> lock(sched_mutex);
  if (not configured) {
    return 0;
  }
  if (enb_cc_idx >= carrier_schedulers.size()) {
    return 0;
  }

  tti_point tti_rx = tti_point{tti_tx_dl} - TX_ENB_DELAY;
  new_tti(tti_rx);

  // copy result
  sched_result = sched_results.get_sf(tti_rx)->get_cc(enb_cc_idx)->dl_sched_result;

  return 0;
}
```

Method Name:

int sched::ul_sched()

Parameters:

uint32_t tti

uint32_t enb_cc_idx

srsenb::sched_interface::ul_sched_res_t& sched_result

Functionality:

Checks upload for configuration

Checks for base station carrier indexing outside of schedulers carrier size

Sets tti receive to tti transmit minus base station transmit delay and HARQ download delay

Sets new tti to tti receive

Get scheduler result

Returns:

0

SRSRAN_SUCCESS

Code:

```
// Uplink Scheduler API
int sched::ul_sched(uint32_t tti, uint32_t enb_cc_idx, srsenb::sched_interface::ul_sched_res_t& sched_result)
{
  std::lock_guard<std::mutex> lock(sched_mutex);
  if (not configured) {
    return 0;
  }
  if (enb_cc_idx >= carrier_schedulers.size()) {
    return 0;
  }

  // Compute scheduling Result for tti_rx
  tti_point tti_rx = tti_point{tti} - TX_ENB_DELAY - FDD_HARQ_DELAY_DL_MS;
  new_tti(tti_rx);

  // copy result
  sched_result = sched_results.get_sf(tti_rx)->get_cc(enb_cc_idx)->ul_sched_result;

  return SRSRAN_SUCCESS;
}
```

Method Name:

void sched::new_tti()

Parameters:

tti_point tti_rx

Functionality:

Sets last tti to tti receive if tti receive is larger

Generates scheduler results for all carriers if not generated

Returns:

void

Code:

```
/// Generate scheduling decision for tti_rx, if it wasn't already generated
/// NOTE: The scheduling decision is made for all CCs in a single call/lock, otherwise the UE can have different
///       configurations (e.g. different set of activated SCells) in different CC decisions
void sched::new_tti(tti_point tti_rx)
{
  last_tti = std::max(last_tti, tti_rx);

  // Generate sched results for all CCs, if not yet generated
  for (size_t cc_idx = 0; cc_idx < carrier_schedulers.size(); ++cc_idx) {
    if (not is_generated(tti_rx, cc_idx)) {
      // Generate carrier scheduling result
      carrier_schedulers[cc_idx]->generate_tti_result(tti_rx);
    }
  }
}
```

Method Name:

bool sched::is_generated() const

Parameters:

srsran::tti_point tti_rx

uint32_t enb_cc_idx

Functionality:

Checks for generated scheduler tti result

Returns:

Scheduler tti result

Code:

```
/// Check if TTI result is generated
bool sched::is_generated(srsran::tti_point tti_rx, uint32_t enb_cc_idx) const
{
  return sched_results.has_sf(tti_rx) and sched_results.get_sf(tti_rx)->is_generated(enb_cc_idx);
}
```

Method Name:

int sched::ue_db_access_locked()

Parameters:

uint16_t rnti

Func&& f

const char* func_name

bool log_fail

Functionality:

Accesses UE database elements in read only fashion

Returns:

SRSRAN_ERROR

SRSRAN_SUCCESS

Code:

```
// Common way to access ue_db elements in a read locking way
template <typename Func>
int sched::ue_db_access_locked(uint16_t rnti, Func&& f, const char* func_name, bool log_fail)
{
  std::lock_guard<std::mutex> lock(sched_mutex);
  auto                        it = ue_db.find(rnti);
  if (it != ue_db.end()) {
    f(*it->second);
  } else {
    if (log_fail) {
      if (func_name != nullptr) {
        Error("SCHED: User rnti=0x%x not found. Failed to call %s.", rnti, func_name);
      } else {
        Error("SCHED: User rnti=0x%x not found.", rnti);
      }
    }
    return SRSRAN_ERROR;
  }
  return SRSRAN_SUCCESS;
}
```

Sched_carrier.cc

Method Name:

void bc_sched::dl_sched()

Parameters:

sf_sched* tti_sched

Functionality:

Sets current tti to tti download transmit

Sets carrier aggregation level to 2

Activates/deactivates system information windows

Allocates DCIs and RBGs for each system information block

Allocates paging

Returns:

void

Code:

```
void bc_sched::dl_sched(sf_sched* tti_sched)
{
  current_tti   = tti_sched->get_tti_tx_dl();
  bc_aggr_level = 2;

  /* Activate/deactivate SI windows */
  update_si_windows(tti_sched);

  /* Allocate DCIs and RBGs for each SIB */
  alloc_sibs(tti_sched);

  /* Allocate Paging */
  // NOTE: It blocks
  alloc_paging(tti_sched);
}
```

Method Name:

void bc_sched::update_si_windows()

Parameters:

sf_sched* tti_sched

Functionality:

Gets tti download transmit

Gets current system function index

Gets current system function number

Checks for system information block data

Adjusts window to ensure SIBs are always in window

Returns:

void

Code:

```
void bc_sched::update_si_windows(sf_sched* tti_sched)
{
  tti_point tti_tx_dl      = tti_sched->get_tti_tx_dl();
  uint32_t  current_sf_idx = tti_sched->get_tti_tx_dl().sf_idx();
  uint32_t  current_sfn    = tti_sched->get_tti_tx_dl().sfn();

  for (uint32_t i = 0; i < pending_sibs.size(); ++i) {
    // There is SIB data
    if (cc_cfg->cfg.sibs[i].len == 0) {
      continue;
    }

    if (not pending_sibs[i].is_in_window) {
      uint32_t sf = 5;
      uint32_t x  = 0;
      if (i > 0) {
        x  = (i - 1) * cc_cfg->cfg.si_window_ms;
        sf = x % 10;
      }
      if ((current_sfn % (cc_cfg->cfg.sibs[i].period_rf)) == x / 10 && current_sf_idx == sf) {
        pending_sibs[i].is_in_window = true;
        pending_sibs[i].window_start = tti_tx_dl;
        pending_sibs[i].n_tx         = 0;
      }
    } else {
      if (i > 0) {
        if (pending_sibs[i].window_start + cc_cfg->cfg.si_window_ms < tti_tx_dl) {
          // the si window has passed
          pending_sibs[i] = {};
        }
      } else {
        // SIB1 is always in window
        if (pending_sibs[0].n_tx == 4) {
          pending_sibs[0].n_tx = 0;
        }


      }
    }
  }
}
```

Method Name:

void bc_sched::alloc_sibs(sf_sched* tti_sched)

Parameters:

sf_sched* tti_sched

Functionality:

Checks that SIBs are in window

Check for subframe index is correct for SIB transmission

Attempt PDSCH grants with increasing number of RBGs

Sets ret accordingly

Returns:

void

Code:

```
void bc_sched::alloc_sibs(sf_sched* tti_sched)
{
  uint32_t current_sf_idx = tti_sched->get_tti_tx_dl().sf_idx();
  uint32_t current_sfn    = tti_sched->get_tti_tx_dl().sfn();

  for (uint32_t sib_idx = 0; sib_idx < pending_sibs.size(); sib_idx++) {
    sched_sib_t& pending_sib = pending_sibs[sib_idx];
    // Check if SIB is configured and within window
    if (cc_cfg->cfg.sibs[sib_idx].len == 0 or not pending_sib.is_in_window or pending_sib.n_tx >= 4) {
      continue;
    }

    // Check if subframe index is the correct one for SIB transmission
    uint32_t nof_tx          = (sib_idx > 0) ? SRSRAN_MIN(srsran::ceil_div(cc_cfg->cfg.si_window_ms, 10), 4) : 4;
    uint32_t n_sf            = (tti_sched->get_tti_tx_dl() - pending_sibs[sib_idx].window_start);
    bool     sib1_flag       = (sib_idx == 0) and (current_sfn % 2) == 0 and current_sf_idx == 5;
    bool     other_sibs_flag = (sib_idx > 0) and
                          (n_sf >= (cc_cfg->cfg.si_window_ms / nof_tx) * pending_sibs[sib_idx].n_tx) and
                          current_sf_idx == 9;
    if (not sib1_flag and not other_sibs_flag) {
      continue;
    }
```

```
    // Attempt PDSCH grants with increasing number of RBGs
    alloc_result ret = alloc_result::invalid_coderate;
    for (uint32_t nrbgs = 1; nrbgs < cc_cfg->nof_rbgs and ret == alloc_result::invalid_coderate; ++nrbgs) {
      rbg_interval rbg_interv = find_empty_rbg_interval(nrbgs, tti_sched->get_dl_mask());
      if (rbg_interv.length() != nrbgs) {
        ret = alloc_result::no_sch_space;
        break;
      }
      ret = tti_sched->alloc_sib(bc_aggr_level, sib_idx, pending_sibs[sib_idx].n_tx, rbg_interv);
      if (ret == alloc_result::success) {
        // SIB scheduled successfully
        pending_sibs[sib_idx].n_tx++;
      }
    }
    if (ret != alloc_result::success) {
      logger.warning("SCHED: Could not allocate SIB=%d, len=%d. Cause: %s",
                     sib_idx + 1,
                     cc_cfg->cfg.sibs[sib_idx].len,
                     to_string(ret));
    }
  }
}
```

Method Name:

void bc_sched::alloc_paging()

Parameters:

sf_sched* tti_sched

Functionality:

Checks for pending paging message

Checks for scheduler space

Adjusts ret

Returns:

void

Code:

```
void bc_sched::alloc_paging(sf_sched* tti_sched)
{
  uint32_t paging_payload = 0;

  // Check if pending Paging message
  if (not rrc->is_paging_opportunity(tti_sched->get_tti_tx_dl().to_uint(), &paging_payload) or paging_payload == 0) {
    return;
  }

  alloc_result ret = alloc_result::invalid_coderate;
  for (uint32_t nrbgs = 1; nrbgs < cc_cfg->nof_rbgs and ret == alloc_result::invalid_coderate; ++nrbgs) {
    rbg_interval rbg_interv = find_empty_rbg_interval(nrbgs, tti_sched->get_dl_mask());
    if (rbg_interv.length() != nrbgs) {
      ret = alloc_result::no_sch_space;
      break;
    }

    ret = tti_sched->alloc_paging(bc_aggr_level, paging_payload, rbg_interv);
  }

  if (ret != alloc_result::success) {
    logger.warning("SCHED: Could not allocate Paging with payload length=%d, cause=%s", paging_payload, to_string(ret));
  }
}
```

Method Name:

void bc_sched::reset()

Parameters:

none

Functionality:

Clears all SIBs

Returns:

void

Code:

```
void bc_sched::reset()
{
  for (auto& sib : pending_sibs) {
    sib = {};
  }
}
```

Method Name:

alloc_result ra_sched::allocate_pending_rar()

Parameters:

sf_sched* tti_sched

const pending_rar_t& rar

uint32_t& nof_grants_alloc

Functionality:

Adjusts ret based on RBGs

Returns:

ret

Code:

```
alloc_result ra_sched::allocate_pending_rar(sf_sched* tti_sched, const pending_rar_t& rar, uint32_t& nof_grants_alloc)
{
  alloc_result ret = alloc_result::other_cause;
  for (nof_grants_alloc = rar.msg3_grant.size(); nof_grants_alloc > 0; nof_grants_alloc--) {
    ret = alloc_result::invalid_coderate;
    for (uint32_t nrbg = 1; nrbg < cc_cfg->nof_rbgs and ret == alloc_result::invalid_coderate; ++nrbg) {
      rbg_interval rbg_interv = find_empty_rbg_interval(nrbg, tti_sched->get_dl_mask());
      if (rbg_interv.length() == nrbg) {
        ret = tti_sched->alloc_rar(rar_aggr_level, rar, rbg_interv, nof_grants_alloc);
      } else {
        ret = alloc_result::no_sch_space;
      }
    }

    // If allocation was not successful because there were not enough RBGs, try allocating fewer Msg3 grants
    if (ret != alloc_result::invalid_coderate and ret != alloc_result::no_sch_space) {
      break;
    }
  }
  if (ret != alloc_result::success) {
    logger.info("SCHED: RAR allocation for L=%d was postponed. Cause=%s", rar_aggr_level, to_string(ret));
  }
  return ret;
}
```

Method Name:

void ra_sched::dl_sched()

Parameters:

sf_sched* tti_sched

Functionality:

Sets aggregation level to 2

Discards RAR for case where it is outside window

Schedules RAR for case where in window

Returns:

void

Code:

```
// Schedules RAR
// On every call to this function, we schedule the oldest RAR which is still within the window. If outside the window we
// discard it.
void ra_sched::dl_sched(sf_sched* tti_sched)
{
  tti_point tti_tx_dl = tti_sched->get_tti_tx_dl();
  rar_aggr_level      = 2;

  for (auto it = pending_rars.begin(); it != pending_rars.end();) {
    auto& rar = *it;

    // In case of RAR outside RAR window:
    // - if window has passed, discard RAR
    // - if window hasn't started, stop loop, as RARs are ordered by TTI
    srsran::tti_interval rar_window{rar.prach_tti + PRACH_RAR_OFFSET,
                                    rar.prach_tti + PRACH_RAR_OFFSET + cc_cfg->cfg.prach_rar_window};
    if (not rar_window.contains(tti_tx_dl)) {
      if (tti_tx_dl >= rar_window.stop()) {
        fmt::memory_buffer str_buffer;
        fmt::format_to(str_buffer,
                       "SCHED: Could not transmit RAR within the window (RA={}, Window={}, RAR={}",
                       rar.prach_tti,
                       rar_window,
                       tti_tx_dl);
        srsran::console("%s\n", srsran::to_c_str(str_buffer));
        logger.warning("%s", srsran::to_c_str(str_buffer));
        it = pending_rars.erase(it);
        continue;
      }
      return;
    }



    // Try to schedule DCI + RBGs for RAR Grant
    uint32_t     nof_rar_allocs = 0;
    alloc_result ret            = allocate_pending_rar(tti_sched, rar, nof_rar_allocs);

    if (ret == alloc_result::success) {
      // If RAR allocation was successful:
      // - in case all Msg3 grants were allocated, remove pending RAR, and continue with following RAR
      // - otherwise, erase only Msg3 grants that were allocated, and stop iteration

      if (nof_rar_allocs == rar.msg3_grant.size()) {
        it = pending_rars.erase(it);
      } else {
        std::copy(rar.msg3_grant.begin() + nof_rar_allocs, rar.msg3_grant.end(), rar.msg3_grant.begin());
        rar.msg3_grant.resize(rar.msg3_grant.size() - nof_rar_allocs);
        break;
      }
    } else {
      // If RAR allocation was not successful:
      // - in case of unavailable PDCCH space, try next pending RAR allocation
      // - otherwise, stop iteration
      if (ret != alloc_result::no_cch_space) {
        break;
      }
      ++it;
    }
  }
}
```

Method Name:

int ra_sched::dl_rach_info()

Parameters:

dl_sched_rar_info_t rar_info

Functionality:

Logs all RAR information

Sets ra_rnti

Looks for pending RAR with same ra_rnti

Returns:

SRSRAN_SUCCESS

SRSRAN_ERROR

Code:

```
int ra_sched::dl_rach_info(dl_sched_rar_info_t rar_info)
{
  logger.info("SCHED: New PRACH tti=%d, preamble=%d, temp_crnti=0x%x, ta_cmd=%d, msg3_size=%d",
              rar_info.prach_tti,
              rar_info.preamble_idx,
              rar_info.temp_crnti,
              rar_info.ta_cmd,
              rar_info.msg3_size);
  // RA-RNTI = 1 + t_id + f_id
  // t_id = index of first subframe specified by PRACH (0<=t_id<10)
  // f_id = index of the PRACH within subframe, in ascending order of freq domain (0<=f_id<6) (for FDD, f_id=0)
  uint16_t ra_rnti = 1 + (uint16_t)(rar_info.prach_tti % 10u);

  // find pending rar with same RA-RNTI
  for (pending_rar_t& r : pending_rars) {
    if (r.prach_tti.to_uint() == rar_info.prach_tti and ra_rnti == r.ra_rnti) {
      if (r.msg3_grant.size() >= sched_interface::MAX_RAR_LIST) {
        logger.warning("PRACH ignored, as the the maximum number of RAR grants per tti has been reached");
        return SRSRAN_ERROR;
      }
      r.msg3_grant.push_back(rar_info);
      return SRSRAN_SUCCESS;
    }
  }

  // create new RAR
  pending_rar_t p;
  p.ra_rnti   = ra_rnti;
  p.prach_tti = tti_point{rar_info.prach_tti};
  p.msg3_grant.push_back(rar_info);
  pending_rars.push_back(p);

  return SRSRAN_SUCCESS;
}
```

Method Name:

void ra_sched::ul_sched()

Parameters:

sf_sched* sf_dl_sched

sf_sched* sf_msg3_sched

Functionality:

Schedules msg3 grants for upload on allocated RARs

Returns:

void

Code:

```
//! Schedule Msg3 grants in UL based on allocated RARs
void ra_sched::ul_sched(sf_sched* sf_dl_sched, sf_sched* sf_msg3_sched)
{
  srsran::const_span<sf_sched::rar_alloc_t> alloc_rars = sf_dl_sched->get_allocated_rars();

  for (const auto& rar : alloc_rars) {
    for (const auto& msg3grant : rar.rar_grant.msg3_grant) {
      uint16_t crnti   = msg3grant.data.temp_crnti;
      auto     user_it = ue_db->find(crnti);
      if (user_it != ue_db->end() and
          sf_msg3_sched->alloc_msg3(user_it->second.get(), msg3grant) == alloc_result::success) {
        logger.debug("SCHED: Queueing Msg3 for rnti=0x%x at tti=%d", crnti, sf_msg3_sched->get_tti_tx_ul().to_uint());
      } else {
        logger.error(
            "SCHED: Failed to allocate Msg3 for rnti=0x%x at tti=%d", crnti, sf_msg3_sched->get_tti_tx_ul().to_uint());
      }
    }
  }
}
```

Method Name:

void ra_sched::reset()

Parameters:

none

Functionality:

Clears pending RARs

Returns:

void

Code:

```
void ra_sched::reset()
{
  pending_rars.clear();
}
```

Method Name:

void sched::carrier_sched::reset()

Parameters:

none

Functionality:

Calls ra and bc sched reset

Returns:

void

Code:

```
void sched::carrier_sched::reset()
{
  ra_sched_ptr.reset();
  bc_sched_ptr.reset();
}
```

Method Name:

void sched::carrier_sched::carrier_cfg()

Parameters:

const sched_cell_params_t& cell_params_

Functionality:

Fully sets carrier schedulers

Initializes bc and ra schedulers

Sets up data scheduling algorithms

Initiates the tti scheduler for each tti

Returns:

void

Code:

```
void sched::carrier_sched::carrier_cfg(const sched_cell_params_t& cell_params_)
{
  // carrier_sched is now fully set
  cc_cfg = &cell_params_;

  // init Broadcast/RA schedulers
  bc_sched_ptr.reset(new bc_sched{*cc_cfg, rrc});
  ra_sched_ptr.reset(new ra_sched{*cc_cfg, *ue_db});

  // Setup data scheduling algorithms
  if (cell_params_.sched_cfg->sched_policy == "time_rr") {
    sched_algo.reset(new sched_time_rr{*cc_cfg, *cell_params_.sched_cfg});
    logger.info("Using time-domain RR scheduling policy for cc=%d", cc_cfg->enb_cc_idx);
  } else {
    sched_algo.reset(new sched_time_pf{*cc_cfg, *cell_params_.sched_cfg});
    logger.info("Using time-domain PF scheduling policy for cc=%d", cc_cfg->enb_cc_idx);
  }

  // Initiate the tti_scheduler for each TTI
  for (sf_sched& tti_sched : sf_scheds) {
    tti_sched.init(*cc_cfg);
  }
}
```

Method Name:

void sched::carrier_sched::set_dl_tti_mask()

Parameters:

uint8_t* tti_mask

uint32_t nof_sfs

Functionality:

Assigns masks to tti

Returns:

void

Code:

```
void sched::carrier_sched::set_dl_tti_mask(uint8_t* tti_mask, uint32_t nof_sfs)
{
  sf_dl_mask.assign(tti_mask, tti_mask + nof_sfs);
}
```

Method Name:

const cc_sched_result& sched::carrier_sched::generate_tti_result()

Parameters:

tti_point tti_rx

Functionality:

Checks for active download

Refreshes UE internal vectors and subframe vars

Schedule PHICH

If download is active:

Schedule broadcast data

Schedule RAR

Schedule msg3

Prioritize PDCCH scheduling for DL and UL data in a RoundRobin fashion

Schedule download user data

Select the winner DCI allocation combination, store all the scheduling results

Reset UE harq pending acknowledge state, clean-up blocked PIDs

Returns:

Cc scheduling results

Code:

```
const cc_sched_result& sched::carrier_sched::generate_tti_result(tti_point tti_rx)
{
  sf_sched*        tti_sched = get_sf_sched(tti_rx);
  sf_sched_result* sf_result = prev_sched_results->get_sf(tti_rx);
  cc_sched_result* cc_result = sf_result->get_cc(enb_cc_idx);

  bool dl_active = sf_dl_mask[tti_sched->get_tti_tx_dl().to_uint() % sf_dl_mask.size()] == 0;

  /* Refresh UE internal buffers and subframe vars */
  for (auto& user : *ue_db) {
    user.second->new_subframe(tti_rx, enb_cc_idx);
  }

  /* Schedule PHICH */
  for (auto& ue_pair : *ue_db) {
    if (tti_sched->alloc_phich(ue_pair.second.get()) == alloc_result::no_grant_space) {
      break;
    }
  }

  /* Schedule DL control data */
  if (dl_active) {
    /* Schedule Broadcast data (SIB and paging) */
    bc_sched_ptr->dl_sched(tti_sched);

    /* Schedule RAR */
    ra_sched_ptr->dl_sched(tti_sched);

    /* Schedule Msg3 */
    sf_sched* sf_msg3_sched = get_sf_sched(tti_rx + MSG3_DELAY_MS);
    ra_sched_ptr->ul_sched(tti_sched, sf_msg3_sched);
  }
```

```
  /* Prioritize PDCCH scheduling for DL and UL data in a RoundRobin fashion */
  if ((tti_rx.to_uint() % 2) == 0) {
    alloc_ul_users(tti_sched);
  }

  /* Schedule DL user data */
  alloc_dl_users(tti_sched);

  if ((tti_rx.to_uint() % 2) == 1) {
    alloc_ul_users(tti_sched);
  }

  /* Select the winner DCI allocation combination, store all the scheduling results */
  tti_sched->generate_sched_results(*ue_db);

  /* Reset ue harq pending ack state, clean-up blocked pids */
  for (auto& user : *ue_db) {
    user.second->finish_tti(tti_rx, enb_cc_idx);
  }

  log_dl_cc_results(logger, enb_cc_idx, cc_result->dl_sched_result);
  log_phich_cc_results(logger, enb_cc_idx, cc_result->ul_sched_result);

  return *cc_result;
}
```

Method Name:

void sched::carrier_sched::alloc_dl_users()

Parameters:

sf_sched* tti_result

Functionality:

Checks for mask

Checks for case of 6 PRBs and does not transmit if PRACH is involved

Calls download scheduler metric to fill RB grid

Returns:

void

Code:

```
void sched::carrier_sched::alloc_dl_users(sf_sched* tti_result)
{
  if (sf_dl_mask[tti_result->get_tti_tx_dl().to_uint() % sf_dl_mask.size()] != 0) {
    return;
  }

  // NOTE: In case of 6 PRBs, do not transmit if there is going to be a PRACH in the UL to avoid collisions
  if (cc_cfg->nof_prb() == 6) {
    tti_point tti_rx_ack = to_tx_dl_ack(tti_result->get_tti_rx());
    if (srsran_prach_tti_opportunity_config_fdd(cc_cfg->cfg.prach_config, tti_rx_ack.to_uint(), -1)) {
      tti_result->reserve_dl_rbgs(0, cc_cfg->nof_rbgs);
    }
  }

  // call DL scheduler metric to fill RB grid
  sched_algo->sched_dl_users(*ue_db, tti_result);
}
```

Method Name:

int sched::carrier_sched::alloc_ul_users()

Parameters:

sf_sched* tti_sched

Functionality:

Calls scheduler for upload data

Returns:

SRSRAN_SUCCESS

Code:

```
int sched::carrier_sched::alloc_ul_users(sf_sched* tti_sched)
{
  /* Call scheduler for UL data */
  sched_algo->sched_ul_users(*ue_db, tti_sched);

  return SRSRAN_SUCCESS;
}
```

Method Name:

sf_sched* sched::carrier_sched::get_sf_sched()

Parameters:

tti_point tti_rx

Functionality:

Assigns ret based on tti receive

Returns:

ret

Code:

```
sf_sched* sched::carrier_sched::get_sf_sched(tti_point tti_rx)
{
  sf_sched* ret = &sf_scheds[tti_rx.to_uint()];
  if (ret->get_tti_rx() != tti_rx) {
    if (not prev_sched_results->has_sf(tti_rx)) {
      // Reset if tti_rx has not been yet set in the sched results
      prev_sched_results->new_tti(tti_rx);
    }
    sf_sched_result* sf_res = prev_sched_results->get_sf(tti_rx);
    // start new TTI for the given CC.
    ret->new_tti(tti_rx, sf_res);
  }
  return ret;
}
```

Method Name:

const sf_sched_result* sched::carrier_sched::get_sf_result() const

Parameters:

tti_point tti_rx

Functionality:

Schedule results from tti receive

Returns:

prev_sched_results->get_sf(tti_rx)

Code:

```
const sf_sched_result* sched::carrier_sched::get_sf_result(tti_point tti_rx) const
{
  return prev_sched_results->get_sf(tti_rx);
}
```

Method Name:

int sched::carrier_sched::dl_rach_info()

Parameters:

dl_sched_rar_info_t rar_info

Functionality:

Returns random access scheduler pointer based on RAR information

Returns:

ra_sched_ptr->dl_rach_info(rar_info)

Code:

```
int sched::carrier_sched::dl_rach_info(dl_sched_rar_info_t rar_info)
{
  return ra_sched_ptr->dl_rach_info(rar_info);
}
```

sched_grid.cc

Method Name:

void sf_sched_result::new_tti()

Parameters:

tti_point tti_rx_

Functionality:

Assigns tti_rx

Clears cc

Returns:

void

Code:

```
void sf_sched_result::new_tti(tti_point tti_rx_)
{
  assert(tti_rx != tti_rx_);
  tti_rx = tti_rx_;
  for (auto& cc : enb_cc_list) {
    cc = {};
  }
}
```

Method Name:

bool sf_sched_result::is_ul_alloc() const

Parameters:

uint16_t rnti

Functionality:

Checks for rnti equalization on download

Returns:

True

False

Code:

```
bool sf_sched_result::is_ul_alloc(uint16_t rnti) const
{
  for (const auto& cc : enb_cc_list) {
    for (const auto& pusch : cc.ul_sched_result.pusch) {
      if (pusch.dci.rnti == rnti) {
        return true;
      }
    }
  }
  return false;
}
```

Method Name:

bool sf_sched_result::is_dl_alloc() const

Parameters:

uint16_t rnti

Functionality:

Checks for RNTI equalization on upload

Returns:

True

False

Code:

```
bool sf_sched_result::is_dl_alloc(uint16_t rnti) const
{
  for (const auto& cc : enb_cc_list) {
    for (const auto& data : cc.dl_sched_result.data) {
      if (data.dci.rnti == rnti) {
        return true;
      }
    }
  }
  return false;
}
```

Method Name:

void sched_result_ringbuffer::set_nof_carriers()

Parameters:

uint32_t nof_carriers_

Functionality:

Sets number of carriers

Resizes base station cc list based on number of carriers

Returns:

void

Code:

```
void sched_result_ringbuffer::set_nof_carriers(uint32_t nof_carriers_)
{
  nof_carriers = nof_carriers_;
  for (auto& sf_res : results) {
    sf_res.enb_cc_list.resize(nof_carriers_);
  }
}
```

Method Name:

void sched_result_ringbuffer::new_tti()

Parameters:

srsran::tti_point tti_rx

Functionality:

Gets res from scheduler results and assigns new tti receive

Returns:

void

Code:

```
void sched_result_ringbuffer::new_tti(srsran::tti_point tti_rx)
{
  sf_sched_result* res = &results[tti_rx.to_uint()];
  res->new_tti(tti_rx);
}
```

Method Name:

void sf_grid_t::init()

Parameters:

const sched_cell_params_t& cell_params_

Functionality:

Resizes masks for download and upload

Sets cc configuration

Sets number of rbgs

Compute reserved PRBs for CQI, SR and HARQ-ACK, and store it in a bitmask

Returns:

void

Code:

```
void sf_grid_t::init(const sched_cell_params_t& cell_params_)
{
  cc_cfg   = &cell_params_;
  nof_rbgs = cc_cfg->nof_rbgs;

  dl_mask.resize(nof_rbgs);
  ul_mask.resize(cc_cfg->nof_prb());

  pdcch_alloc.init(*cc_cfg);

  // Compute reserved PRBs for CQI, SR and HARQ-ACK, and store it in a bitmask
  pucch_mask.resize(cc_cfg->nof_prb());
  pucch_nrb                    = (cc_cfg->cfg.nrb_pucch > 0) ? (uint32_t)cc_cfg->cfg.nrb_pucch : 0;
  srsran_pucch_cfg_t pucch_cfg = cell_params_.pucch_cfg_common;
  pucch_cfg.n_pucch            = cc_cfg->nof_cce_table[SRSRAN_NOF_CFI - 1] - 1 + cc_cfg->cfg.n1pucch_an;
  pucch_nrb                    = std::max(pucch_nrb, srsran_pucch_m(&pucch_cfg, cc_cfg->cfg.cell.cp) / 2 + 1);
  if (pucch_nrb > 0) {
    pucch_mask.fill(0, pucch_nrb);
    pucch_mask.fill(cc_cfg->nof_prb() - pucch_nrb, cc_cfg->nof_prb());
  }
}
```

Method Name:

void sf_grid_t::new_tti()

Parameters:

tti_point tti_rx_

Functionality:

Resets masks

Reserve PRBs for PUCCH

Reserve PRBs for PRACH

Allocates tti for PDCCH

Returns:

void

Code:

```
void sf_grid_t::new_tti(tti_point tti_rx_)
{
  tti_rx = tti_rx_;

  dl_mask.reset();
  ul_mask.reset();

  // Reserve PRBs for PUCCH
  ul_mask |= pucch_mask;

  // Reserve PRBs for PRACH
  if (srsran_prach_tti_opportunity_config_fdd(cc_cfg->cfg.prach_config, to_tx_ul(tti_rx).to_uint(), -1)) {
    prbmask_t prach_mask{cc_cfg->nof_prb()};
    prach_mask.fill(cc_cfg->cfg.prach_freq_offset, cc_cfg->cfg.prach_freq_offset + 6);
    reserve_ul_prbs(prach_mask, false); // TODO: set to true once test sib.conf files are updated
    if (logger.debug.enabled()) {
      fmt::memory_buffer buffer;
      fmt::format_to(buffer, "SCHED: Allocated PRACH RBs mask={:x} for tti_tx_ul={}", prach_mask, to_tx_ul(tti_rx));
      logger.debug("%s", srsran::to_c_str(buffer));
    }
  }

  // internal state
  pdcch_alloc.new_tti(tti_rx);
}
```

Method Name:

alloc_result sf_grid_t::alloc_dl()

Parameters:

uint32_t aggr_idx

alloc_type_t alloc_type

rbgmask_t alloc_mask

sched_ue* user

bool has_pusch_grant

Functionality:

Allocates CCEs and RBs for the given mask and allocation type

Checks RBG collision

Allocate DCI in PDCCH

Allocates RBGs

Returns:

alloc_result::success

alloc_result::no_cch_space

alloc_result::sch_collision

Code:

```
alloc_result sf_grid_t::alloc_dl(uint32_t     aggr_idx,
                                 alloc_type_t alloc_type,
                                 rbgmask_t    alloc_mask,
                                 sched_ue*    user,
                                 bool         has_pusch_grant)
{
  // Check RBG collision
  if ((dl_mask & alloc_mask).any()) {
    logger.debug("SCHED: Provided RBG mask collides with allocation previously made.\n");
    return alloc_result::sch_collision;
  }

  // Allocate DCI in PDCCH
  if (not pdcch_alloc.alloc_dci(alloc_type, aggr_idx, user, has_pusch_grant)) {
    if (logger.debug.enabled()) {
      if (user != nullptr) {
        logger.debug("SCHED: No space in PDCCH for rnti=0x%x DL tx. Current PDCCH allocation:\n%s",
                     user->get_rnti(),
                     pdcch_alloc.result_to_string(true).c_str());
      } else {
        logger.debug("SCHED: No space in PDCCH for DL tx. Current PDCCH allocation:\n%s",
                     pdcch_alloc.result_to_string(true).c_str());
      }
    }
    return alloc_result::no_cch_space;
  }

  // Allocate RBGs
  dl_mask |= alloc_mask;

  return alloc_result::success;
}
```

Method Name:

alloc_result sf_grid_t::alloc_dl_ctrl()

Parameters:

uint32_t aggr_idx

rbg_interval rbg_range

alloc_type_t alloc_type

Functionality:

Allocates CCEs and RBs for control allocs

It allocates RBs in a continuous manner

Setup rbg_range starting from left

allocate DCI and RBGs

Returns:

alloc_result::other_cause

alloc_result::sch_collision

alloc_dl(aggr_idx, alloc_type, new_mask)

Code:

```
/// Allocates CCEs and RBs for control allocs. It allocates RBs in a contiguous manner.
alloc_result sf_grid_t::alloc_dl_ctrl(uint32_t aggr_idx, rbg_interval rbg_range, alloc_type_t alloc_type)
{
  if (alloc_type != alloc_type_t::DL_RAR and alloc_type != alloc_type_t::DL_BC and
      alloc_type != alloc_type_t::DL_PCCH) {
    logger.error("SCHED: DL control allocations must be RAR/BC/PDCCH");
    return alloc_result::other_cause;
  }
  // Setup rbg_range starting from left
  if (rbg_range.stop() > nof_rbgs) {
    return alloc_result::sch_collision;
  }

  // allocate DCI and RBGs
  rbgmask_t new_mask(dl_mask.size());
  new_mask.fill(rbg_range.start(), rbg_range.stop());
  return alloc_dl(aggr_idx, alloc_type, new_mask);
}
```

Method Name:

alloc_result sf_grid_t::alloc_dl_data()

Parameters:

sched_ue* user

const rbgmask_t& user_mask

bool has_pusch_grant

Functionality:

Allocates CCEs and RBs for a user DL data alloc

Returns:

ret

Code:

```
alloc_result sf_grid_t::alloc_dl_data(sched_ue* user, const rbgmask_t& user_mask, bool has_pusch_grant)
{
  srsran_dci_format_t dci_format = user->get_dci_format();
  uint32_t            nof_bits   = srsran_dci_format_sizeof(&cc_cfg->cfg.cell, nullptr, nullptr, dci_format);
  uint32_t            aggr_idx   = user->get_aggr_level(cc_cfg->enb_cc_idx, nof_bits);
  alloc_result        ret        = alloc_dl(aggr_idx, alloc_type_t::DL_DATA, user_mask, user, has_pusch_grant);

  return ret;
}
```

Method Name:

alloc_result sf_grid_t::alloc_ul_data()

Parameters:

sched_ue* user

prb_interval alloc

bool needs_pdcch

bool strict

Functionality:

Checks not exceeding mask length

Checks for collision

Generate PDCCH except for RAR and non-adaptive retx

Sets upload mask

Returns:

alloc_result::no_sch_space

alloc_result::sch_collision

alloc_result::no_cch_space

alloc_result::success

Code:

```
alloc_result sf_grid_t::alloc_ul_data(sched_ue* user, prb_interval alloc, bool needs_pdcch, bool strict)
{
  if (alloc.stop() > ul_mask.size()) {
    return alloc_result::no_sch_space;
  }

  prbmask_t newmask(ul_mask.size());
  newmask.fill(alloc.start(), alloc.stop());
  if (strict and (ul_mask & newmask).any()) {
    logger.debug("SCHED: Failed UL allocation. Cause: %s", to_string(alloc_result::sch_collision));
    return alloc_result::sch_collision;
  }

  // Generate PDCCH except for RAR and non-adaptive retx
  if (needs_pdcch) {
    uint32_t nof_bits = srsran_dci_format_sizeof(&cc_cfg->cfg.cell, nullptr, nullptr, SRSRAN_DCI_FORMAT0);
    uint32_t aggr_idx = user->get_aggr_level(cc_cfg->enb_cc_idx, nof_bits);
    if (not pdcch_alloc.alloc_dci(alloc_type_t::UL_DATA, aggr_idx, user)) {
      if (logger.debug.enabled()) {
        logger.debug("No space in PDCCH for rnti=0x%x UL tx. Current PDCCH allocation:\n%s",
                     user->get_rnti(),
                     pdcch_alloc.result_to_string(true).c_str());
      }
      return alloc_result::no_cch_space;
    }
  }

  ul_mask |= newmask;

  return alloc_result::success;
}
```

Method Name:

bool sf_grid_t::reserve_dl_rbgs()

Parameters:

uint32_t start_rbg

uint32_t end_rbg

Functionality:

Fills download mask

Returns:

True

Code:

```
bool sf_grid_t::reserve_dl_rbgs(uint32_t start_rbg, uint32_t end_rbg)
{
  dl_mask.fill(start_rbg, end_rbg);
  return true;
}
```

Method Name:

void sf_grid_t::rem_last_alloc_dl()

Parameters:

rbg_interval rbgs

Functionality:

Checks for empty grid

Sets download mask

Returns:

void

Code:

```
void sf_grid_t::rem_last_alloc_dl(rbg_interval rbgs)
{
  if (pdcch_alloc.nof_allocs() == 0) {
    logger.error("Remove DL alloc called for empty Subframe RB grid");
    return;
  }

  pdcch_alloc.rem_last_dci();
  rbgmask_t rbgmask(dl_mask.size());
  rbgmask.fill(rbgs.start(), rbgs.stop());
  dl_mask &= ~rbgmask;
}
```

Method Name:

alloc_result sf_grid_t::reserve_ul_prbs()

Parameters:

prb_interval alloc

bool strict

Functionality:

Checks upload mask size

Fills new mask for upload

Returns:

alloc_result::invalid_grant_params

reserve_ul_prbs(newmask, strict)

Code:

```
alloc_result sf_grid_t::reserve_ul_prbs(prb_interval alloc, bool strict)
{
  if (alloc.stop() > ul_mask.size()) {
    return alloc_result::invalid_grant_params;
  }

  prbmask_t newmask(ul_mask.size());
  newmask.fill(alloc.start(), alloc.stop());
  return reserve_ul_prbs(newmask, strict);
}
```

Method Name:

alloc_result sf_grid_t::reserve_ul_prbs()

Parameters:

const prbmask_t& prbmask

bool strict

Functionality:

Sets ret

Sets upload mask

Returns:

ret

Code:

```
alloc_result sf_grid_t::reserve_ul_prbs(const prbmask_t& prbmask, bool strict)
{
  alloc_result ret = alloc_result::success;
  if (strict and (ul_mask & prbmask).any()) {
    if (logger.info.enabled()) {
      fmt::memory_buffer tmp_buffer;
      fmt::format_to(
          tmp_buffer, "There was a collision in the UL. Current mask=0x{:x}, new mask=0x{:x}", ul_mask, prbmask);
      logger.info("%s", srsran::to_c_str(tmp_buffer));
      ret = alloc_result::sch_collision;
    }
  }
  ul_mask |= prbmask;
  return ret;
}
```

Method Name:

bool sf_grid_t::find_ul_alloc() const

Parameters:

uint32_t L

prb_interval* alloc

Functionality:

Finds a range of L contiguous PRBs that are empty

L Size of the requested UL allocation in PRBs

alloc Found allocation. It is guaranteed that 0 <= alloc->L <= L

Returns true if the requested allocation of size L was strictly met

Returns:

True

False

Code:

```
bool sf_grid_t::find_ul_alloc(uint32_t L, prb_interval* alloc) const
{
  *alloc = {};
  for (uint32_t n = 0; n < ul_mask.size() && alloc->length() < L; n++) {
    if (not ul_mask.test(n) && alloc->length() == 0) {
      alloc->displace_to(n);
    }
    if (not ul_mask.test(n)) {
      alloc->resize_by(1);
    } else if (alloc->length() > 0) {
      // avoid edges
      if (n < 3) {
        *alloc = {};
      } else {
        break;
      }
    }
  }
  if (alloc->length() == 0) {
    return false;
  }

  // Make sure L is allowed by SC-FDMA modulation
  while (!srsran_dft_precoding_valid_prb(alloc->length())) {
    alloc->resize_by(-1);
  }
  return alloc->length() == L;
}
```

Method Name:

void sf_sched::init()

Parameters:

const sched_cell_params_t& cell_params_

Functionality:

Sets cc configuration

Sets max for msg3 PRBs

Returns:

void

Code:

```
void sf_sched::init(const sched_cell_params_t& cell_params_)
{
  cc_cfg = &cell_params_;
  tti_alloc.init(*cc_cfg);
  max_msg3_prb = std::max(6U, cc_cfg->cfg.cell.nof_prb - tti_alloc.get_pucch_width());
}
```

Method Name:

void sf_sched::new_tti()

Parameters:

tti_point tti_rx_

sf_sched_result* cc_results_

Functionality:

Resets internal state

Setup first prb to be used for msg3 allocation

Account for potential PRACH allocation

Returns:

void

Code:

```
void sf_sched::new_tti(tti_point tti_rx_, sf_sched_result* cc_results_)
{
  // reset internal state
  bc_allocs.clear();
  rar_allocs.clear();
  data_allocs.clear();
  ul_data_allocs.clear();

  tti_rx = tti_rx_;
  tti_alloc.new_tti(tti_rx_);
  cc_results = cc_results_;

  // setup first prb to be used for msg3 alloc. Account for potential PRACH alloc
  last_msg3_prb            = tti_alloc.get_pucch_width();
  tti_point tti_msg3_alloc = to_tx_ul(tti_rx) + MSG3_DELAY_MS;
  if (srsran_prach_tti_opportunity_config_fdd(cc_cfg->cfg.prach_config, tti_msg3_alloc.to_uint(), -1)) {
    last_msg3_prb = std::max(last_msg3_prb, cc_cfg->cfg.prach_freq_offset + 6);
  }
}
```

Method Name:

bool sf_sched::is_dl_alloc() const

Parameters:

uint16_t rnti

Functionality:

Returns and download allocation information

Returns:

std::any_of(data_allocs.begin(), data_allocs.end(), [rnti](const dl_alloc_t& u) {

return u.rnti == rnti; })

Code:

```
bool sf_sched::is_dl_alloc(uint16_t rnti) const
{
  return std::any_of(data_allocs.begin(), data_allocs.end(), [rnti](const dl_alloc_t& u) { return u.rnti == rnti; });
}
```

Method Name:

bool sf_sched::is_ul_alloc() const

Parameters:

uint16_t rnti

Functionality:

Returns and upload allocation information

Returns:

std::any_of(ul_data_allocs.begin(), ul_data_allocs.end(), [rnti](const ul_alloc_t&

u) { return u.rnti == rnti; })

Code:

```
bool sf_sched::is_ul_alloc(uint16_t rnti) const
{
  return std::any_of(
      ul_data_allocs.begin(), ul_data_allocs.end(), [rnti](const ul_alloc_t& u) { return u.rnti == rnti; });
}
```

Method Name:

alloc_result sf_sched::alloc_sib()

Parameters:

uint32_t aggr_lvl

uint32_t sib_idx

uint32_t sib_ntx

rbg_interval rbgs

Functionality:

Checks for full broadcast allocations

Allocate SIB RBGs and PDCCH

Generate DCI for SIB

Allocates SIBs

Returns:

Alloc_result::no_grant_space

Ret

Alloc_result::invalid_coderate

alloc_result::success

Code:

```
alloc_result sf_sched::alloc_sib(uint32_t aggr_lvl, uint32_t sib_idx, uint32_t sib_ntx, rbg_interval rbgs)
{
  if (bc_allocs.full()) {
    logger.warning("SCHED: Maximum number of Broadcast allocations reached");
    return alloc_result::no_grant_space;
  }
  bc_alloc_t bc_alloc;

  // Allocate SIB RBGs and PDCCH
  alloc_result ret = tti_alloc.alloc_dl_ctrl(aggr_lvl, rbgs, alloc_type_t::DL_BC);
  if (ret != alloc_result::success) {
    return ret;
  }

  // Generate DCI for SIB
  if (not generate_sib_dci(bc_alloc.bc_grant, get_tti_tx_dl(), sib_idx, sib_ntx, rbgs, *cc_cfg, tti_alloc.get_cfi())) {
    // Cancel on-going allocation
    tti_alloc.rem_last_alloc_dl(rbgs);
    return alloc_result::invalid_coderate;
  }

  // Allocation Successful
  bc_alloc.dci_idx   = tti_alloc.get_pdcch_grid().nof_allocs() - 1;
  bc_alloc.rbg_range = rbgs;
  bc_alloc.req_bytes = cc_cfg->cfg.sibs[sib_idx].len;
  bc_allocs.push_back(bc_alloc);

  return alloc_result::success;
}
```

Method Name:

alloc_result sf_sched::alloc_paging()

Parameters:

uint32_t aggr_lvl

uint32_t paging_payload

rbg_interval rbgs

Functionality:

Checks for full broadcast allocations

Allocates paging RBGs and PDCCH

Generates DCI for Paging message

Allocates paging

Returns:

Alloc_result::no_grant_space

Ret

Alloc_result::invalid_coderate

alloc_result::success

Code:

```
alloc_result sf_sched::alloc_paging(uint32_t aggr_lvl, uint32_t paging_payload, rbg_interval rbgs)
{
  if (bc_allocs.full()) {
    logger.warning("SCHED: Maximum number of Broadcast allocations reached");
    return alloc_result::no_grant_space;
  }
  bc_alloc_t bc_alloc;

  // Allocate Paging RBGs and PDCCH
  alloc_result ret = tti_alloc.alloc_dl_ctrl(aggr_lvl, rbgs, alloc_type_t::DL_PCCH);
  if (ret != alloc_result::success) {
    return ret;
  }

  // Generate DCI for Paging message
  if (not generate_paging_dci(bc_alloc.bc_grant, get_tti_tx_dl(), paging_payload, rbgs, *cc_cfg, tti_alloc.get_cfi())) {
    // Cancel on-going allocation
    tti_alloc.rem_last_alloc_dl(rbgs);
    return alloc_result::invalid_coderate;
  }

  // Allocation Successful
  bc_alloc.dci_idx   = tti_alloc.get_pdcch_grid().nof_allocs() - 1;
  bc_alloc.rbg_range = rbgs;
  bc_alloc.req_bytes = paging_payload;
  bc_allocs.push_back(bc_alloc);

  return alloc_result::success;
}
```

Method Name:

alloc_result sf_sched::alloc_rar()

Parameters:

uint32_t aggr_lvl

const pending_rar_t& rar

rbg_interval rbgs

uint32_t nof_grants

Functionality:

Sets number of PRBs for msg3

Checks for max RAR allocations

Sets RAR buffer

Sets total number of PRBs for upload

Checks if there is enough space for Msg3

Allocates RBGs and PDCCH

Generates DCI for RAR

Allocates RAR

Returns:

Alloc_result::no_grant_space

Ret

Alloc_result::invalid_coderate

alloc_result::sch_collision

Code:

```
alloc_result sf_sched::alloc_rar(uint32_t aggr_lvl, const pending_rar_t& rar, rbg_interval rbgs, uint32_t nof_grants)
{
  static const uint32_t msg3_nof_prbs = 3;
  if (rar_allocs.full()) {
    logger.info("SCHED: Maximum number of RAR allocations per TTI reached.");
    return alloc_result::no_grant_space;
  }

  uint32_t buf_rar           = 7 * nof_grants + 1; // 1+6 bytes per RAR subheader+body and 1 byte for Backoff
  uint32_t total_ul_nof_prbs = msg3_nof_prbs * nof_grants;

  // check if there is enough space for Msg3
  if (last_msg3_prb + total_ul_nof_prbs > max_msg3_prb) {
    return alloc_result::sch_collision;
  }

  // allocate RBGs and PDCCH
  alloc_result ret = tti_alloc.alloc_dl_ctrl(aggr_lvl, rbgs, alloc_type_t::DL_RAR);
  if (ret != alloc_result::success) {
    return ret;
  }

  // Generate DCI for RAR
  rar_alloc_t rar_alloc;
  if (not generate_rar_dci(
          rar_alloc.rar_grant, get_tti_tx_dl(), rar, rbgs, nof_grants, last_msg3_prb, *cc_cfg, tti_alloc.get_cfi())) {
    // Cancel on-going allocation
    tti_alloc.rem_last_alloc_dl(rbgs);
    return alloc_result::invalid_coderate;
  }



  // RAR allocation successful
  rar_alloc.alloc_data.dci_idx   = tti_alloc.get_pdcch_grid().nof_allocs() - 1;
  rar_alloc.alloc_data.rbg_range = rbgs;
  rar_alloc.alloc_data.req_bytes = buf_rar;
  rar_allocs.push_back(rar_alloc);
  last_msg3_prb += total_ul_nof_prbs * nof_grants;

  return ret;
}
```

Method Name:

bool is_periodic_cqi_expected()

Parameters:

const sched_interface::ue_cfg_t& ue_cfg

tti_point tti_tx_ul

Functionality:

Checks for expected cqi

Returns:

True

False

Code:

```
bool is_periodic_cqi_expected(const sched_interface::ue_cfg_t& ue_cfg, tti_point tti_tx_ul)
{
  for (const sched_interface::ue_cfg_t::cc_cfg_t& cc : ue_cfg.supported_cc_list) {
    if (cc.dl_cfg.cqi_report.periodic_configured) {
      if (srsran_cqi_periodic_send(&cc.dl_cfg.cqi_report, tti_tx_ul.to_uint(), SRSRAN_FDD)) {
        return true;
      }
    }
  }
  return false;
}
```

Method Name:

alloc_result sf_sched::alloc_dl_user()

Parameters:

sched_ue* user

const rbgmask_t& user_mask

uint32_t pid

Functionality:

Checks for full download allocation data

Checks for not assigning multiple harqs to one user

Checks for UE carrier

Checks for PDSCH enabled on UE

Check if allocation would cause segmentation

Checks for allocation for DCI format

Checks for PUSCH grant

Check if there is space in the PUCCH for HARQ ACKs

Try to allocate RBGs, PDCCH, and PUCCH

Returns:

Alloc_result::no_grant_space

Alloc_result::no_rnti_opportunity

Alloc_result::invalid_grant_params

Alloc_result::no_cch_space

Ret

alloc_result::success

Code:

```
alloc_result sf_sched::alloc_dl_user(sched_ue* user, const rbgmask_t& user_mask, uint32_t pid)
{
  if (data_allocs.full()) {
    logger.warning("SCHED: Maximum number of DL allocations reached");
    return alloc_result::no_grant_space;
  }

  if (is_dl_alloc(user->get_rnti())) {
    logger.warning("SCHED: Attempt to assign multiple harq pids to the same user rnti=0x%x", user->get_rnti());
    return alloc_result::no_rnti_opportunity;
  }

  auto* cc = user->find_ue_carrier(cc_cfg->enb_cc_idx);
  if (cc == nullptr or cc->cc_state() != cc_st::active) {
    return alloc_result::no_rnti_opportunity;
  }
  if (not user->pdsch_enabled(srsran::tti_point{get_tti_rx()}, cc_cfg->enb_cc_idx)) {
    return alloc_result::no_rnti_opportunity;
  }

  // Check if allocation would cause segmentation
  const dl_harq_proc& h = user->get_dl_harq(pid, cc_cfg->enb_cc_idx);
  if (h.is_empty()) {
    // It is newTx
    rbg_interval r = user->get_required_dl_rbgs(cc_cfg->enb_cc_idx);
    if (r.start() > user_mask.count()) {
      logger.debug("SCHED: The number of RBGs allocated to rnti=0x%x will force segmentation", user->get_rnti());
      return alloc_result::invalid_grant_params;
    }
  }

  srsran_dci_format_t dci_format = user->get_dci_format();
  if (dci_format == SRSRAN_DCI_FORMAT1A and not is_contiguous(user_mask)) {
    logger.warning("SCHED: Can't use distributed RBGs for DCI format 1A");
    return alloc_result::invalid_grant_params;
  }
```

```
bool has_pusch_grant = is_ul_alloc(user->get_rnti()) or cc_results->is_ul_alloc(user->get_rnti());

// Check if there is space in the PUCCH for HARQ ACKs
const sched_interface::ue_cfg_t& ue_cfg    = user->get_ue_cfg();
std::bitset<SRSRAN_MAX_CARRIERS> scells    = user->scell_activation_mask();
uint32_t                         ue_cc_idx = cc->get_ue_cc_idx();
if (user->nof_carriers_configured() > 1 and (ue_cc_idx == 0 or scells[ue_cc_idx]) and
    is_periodic_cqi_expected(ue_cfg, get_tti_tx_ul()) and not has_pusch_grant and
    user->get_ul_harq(get_tti_tx_ul(), get_enb_cc_idx())->is_empty()) {
  // Try to allocate small PUSCH grant, if there are no allocated PUSCH grants for this TTI yet
  prb_interval alloc = {};
  uint32_t L = user->get_required_prb_ul(cc_cfg->enb_cc_idx, srsran::ceil_div(SRSRAN_UCI_CQI_CODED_PUCCH_B + 2, 8));
  tti_alloc.find_ul_alloc(L, &alloc);
  has_pusch_grant = alloc.length() > 0 and alloc_ul_user(user, alloc) == alloc_result::success;
  if (ue_cc_idx != 0 and not has_pusch_grant) {
    // For SCells, if we can't allocate small PUSCH grant, abort DL allocation
    return alloc_result::no_cch_space;
  }
}



  // Try to allocate RBGs, PDCCH, and PUCCH
  alloc_result ret = tti_alloc.alloc_dl_data(user, user_mask, has_pusch_grant);

  if (ret == alloc_result::no_cch_space and not has_pusch_grant and not data_allocs.empty() and
      user->get_ul_harq(get_tti_tx_ul(), get_enb_cc_idx())->is_empty()) {
    // PUCCH may be too full. Attempt small UL grant allocation for UCI-PUSCH
    uint32_t L = user->get_required_prb_ul(cc_cfg->enb_cc_idx, srsran::ceil_div(SRSRAN_UCI_CQI_CODED_PUCCH_B + 2, 8));
    prb_interval alloc = {};
    tti_alloc.find_ul_alloc(L, &alloc);
    has_pusch_grant = alloc.length() > 0 and alloc_ul_user(user, alloc) == alloc_result::success;
    if (has_pusch_grant) {
      ret = tti_alloc.alloc_dl_data(user, user_mask, has_pusch_grant);
    }
  }
  if (ret != alloc_result::success) {
    return ret;
  }

  // Allocation Successful
  dl_alloc_t alloc;
  alloc.dci_idx   = tti_alloc.get_pdcch_grid().nof_allocs() - 1;
  alloc.rnti      = user->get_rnti();
  alloc.user_mask = user_mask;
  alloc.pid       = pid;
  data_allocs.push_back(alloc);

  return alloc_result::success;
}
```

Method Name:

alloc_result sf_sched::alloc_ul()

Parameters:

sched_ue* user

prb_interval alloc

ul_alloc_t::type_t alloc_type

bool is_msg3

int msg3_mcs

Functionality:

Check for max upload allocations

Checks for multiples upload grants case

Check for collision with measGap

Allocate RBGs and DCI space

Returns:

Alloc_result::no_grant_space

Alloc_result::no_rnti_opportunity

Alloc_result::invalid_grant_params

Ret

alloc_result::success

Code:

```
alloc_result
sf_sched::alloc_ul(sched_ue* user, prb_interval alloc, ul_alloc_t::type_t alloc_type, bool is_msg3, int msg3_mcs)
{
  if (ul_data_allocs.full()) {
    logger.debug("SCHED: Maximum number of UL allocations=%zd reached", ul_data_allocs.size());
    return alloc_result::no_grant_space;
  }

  if (is_ul_alloc(user->get_rnti())) {
    logger.warning("SCHED: Attempt to assign multiple UL grants to the same user rnti=0x%x", user->get_rnti());
    return alloc_result::no_rnti_opportunity;
  }

  // Check if there is no collision with measGap
  bool needs_pdcch = alloc_type == ul_alloc_t::ADAPT_RETX or (alloc_type == ul_alloc_t::NEWTX and not is_msg3);
  if (not user->pusch_enabled(get_tti_rx(), cc_cfg->enb_cc_idx, needs_pdcch)) {
    logger.debug("SCHED: PDCCH would collide with rnti=0x%x Measurement Gap", user->get_rnti());
    return alloc_result::no_rnti_opportunity;
  }

  // Allocate RBGs and DCI space
  bool         allow_pucch_collision = cc_cfg->nof_prb() == 6 and is_msg3;
  alloc_result ret                   = tti_alloc.alloc_ul_data(user, alloc, needs_pdcch, not allow_pucch_collision);
  if (ret != alloc_result::success) {
    return ret;
  }

  ul_data_allocs.emplace_back();
  ul_alloc_t& ul_alloc = ul_data_allocs.back();
  ul_alloc.type        = alloc_type;
  ul_alloc.is_msg3     = is_msg3;
  ul_alloc.dci_idx     = tti_alloc.get_pdcch_grid().nof_allocs() - 1;
  ul_alloc.rnti        = user->get_rnti();
  ul_alloc.alloc       = alloc;
  ul_alloc.msg3_mcs    = msg3_mcs;

  return alloc_result::success;
}
```

Method Name:

alloc_result sf_sched::alloc_ul_user()

Parameters:

sched_ue* user

prb_interval alloc

Functionality:

check whether adaptive/non-adaptive retx/newtx

Returns:

alloc_ul(user, alloc, alloc_type, h->is_msg3())

Code:

```
alloc_result sf_sched::alloc_ul_user(sched_ue* user, prb_interval alloc)
{
  // check whether adaptive/non-adaptive retx/newtx
  ul_alloc_t::type_t alloc_type;
  ul_harq_proc*      h        = user->get_ul_harq(get_tti_tx_ul(), cc_cfg->enb_cc_idx);
  bool               has_retx = h->has_pending_retx();
  if (not has_retx) {
    alloc_type = ul_alloc_t::NEWTX;
  } else if (h->retx_requires_pdcch(get_tti_tx_ul(), alloc)) {
    alloc_type = ul_alloc_t::ADAPT_RETX;
  } else {
    alloc_type = ul_alloc_t::NOADAPT_RETX;
  }
```

Method Name:

alloc_result sf_sched::alloc_phich()

Parameters:

sched_ue* user

Functionality:

Set up PHICH

Check for max PHICH allocations

Check user supports carrier

Assign HARQ

Indicate PHICH acknowledgment if needed

Returns:

Alloc_result::no_grant_space

Alloc_result::no_rnti_opportunity

alloc_result::success

Code:

```
alloc_result sf_sched::alloc_phich(sched_ue* user)
{
  using phich_t = sched_interface::ul_sched_phich_t;

  auto* ul_sf_result = &cc_results->get_cc(cc_cfg->enb_cc_idx)->ul_sched_result;
  if (ul_sf_result->phich.full()) {
    logger.warning("SCHED: Maximum number of PHICH allocations has been reached");
    return alloc_result::no_grant_space;
  }

  auto p = user->get_active_cell_index(cc_cfg->enb_cc_idx);
  if (not p.first) {
    // user does not support this carrier
    return alloc_result::no_rnti_opportunity;
  }

  ul_harq_proc* h = user->get_ul_harq(get_tti_tx_ul(), cc_cfg->enb_cc_idx);

  /* Indicate PHICH acknowledgment if needed */
  if (h->has_pending_phich()) {
    ul_sf_result->phich.emplace_back();
    ul_sf_result->phich.back().rnti  = user->get_rnti();
    ul_sf_result->phich.back().phich = h->pop_pending_phich() ? phich_t::ACK : phich_t::NACK;
    return alloc_result::success;
  }
  return alloc_result::no_rnti_opportunity;
}
```

Method Name:

void sf_sched::set_dl_data_sched_result()

Parameters:

const sf_cch_allocator::alloc_result_t& dci_result,

sched_interface::dl_sched_res_t* dl_result

sched_ue_list& ue_list

Functionality:

Assign NCCE/L

Generate DCI Format1/2/2A

Assigns user data

Generates user download DCI

Prints Resulting DL Allocation

Returns:

void

Code:

```
void sf_sched::set_dl_data_sched_result(const sf_cch_allocator::alloc_result_t& dci_result,
                                        sched_interface::dl_sched_res_t*        dl_result,
                                        sched_ue_list&                          ue_list)
{
  for (const auto& data_alloc : data_allocs) {
    dl_result->data.emplace_back();
    sched_interface::dl_sched_data_t* data = &dl_result->data.back();

    // Assign NCCE/L
    data->dci.location = dci_result[data_alloc.dci_idx]->dci_pos;

    // Generate DCI Format1/2/2A
    auto ue_it = ue_list.find(data_alloc.rnti);
    if (ue_it == ue_list.end()) {
      continue;
    }
    sched_ue*           user        = ue_it->second.get();
    uint32_t            data_before = user->get_pending_dl_bytes(cc_cfg->enb_cc_idx);
    const dl_harq_proc& dl_harq     = user->get_dl_harq(data_alloc.pid, cc_cfg->enb_cc_idx);
    bool                is_newtx    = dl_harq.is_empty();

    int tbs = user->generate_dl_dci_format(
        data_alloc.pid, data, get_tti_tx_dl(), cc_cfg->enb_cc_idx, tti_alloc.get_cfi(), data_alloc.user_mask);

    if (tbs <= 0) {
      fmt::memory_buffer str_buffer;
      fmt::format_to(str_buffer,
                     "SCHED: DL {} failed rnti=0x{:x}, pid={}, mask={:x}, tbs={}, buffer={}",
                     is_newtx ? "tx" : "retx",
                     user->get_rnti(),
                     data_alloc.pid,
                     data_alloc.user_mask,
                     tbs,
                     user->get_pending_dl_bytes(cc_cfg->enb_cc_idx));
      logger.warning("%s", srsran::to_c_str(str_buffer));
      continue;
    }


    // Print Resulting DL Allocation
    fmt::memory_buffer str_buffer;
    fmt::format_to(str_buffer,
                   "SCHED: DL {} rnti=0x{:x}, cc={}, pid={}, mask=0x{:x}, dci=({}, {}), n_rtx={}, tbs={}, "
                   "buffer={}/{}, tti_tx_dl={}",
                   is_newtx ? "tx" : "retx",
                   user->get_rnti(),
                   cc_cfg->enb_cc_idx,
                   data_alloc.pid,
                   data_alloc.user_mask,
                   data->dci.location.L,
                   data->dci.location.ncce,
                   dl_harq.nof_retx(0) + dl_harq.nof_retx(1),
                   tbs,
                   data_before,
                   user->get_pending_dl_bytes(cc_cfg->enb_cc_idx),
                   get_tti_tx_dl());
    logger.info("%s", srsran::to_c_str(str_buffer));
  }
}
```

Method Name:

uci_pusch_t is_uci_included()

Parameters:

const sf_sched* sf_sched

const sf_sched_result& other_cc_results

const sched_ue* user

uint32_t current_enb_cc_idx

Functionality:

Finds eNB CC Idex that currently holds UCI

Check if UCI needs to be allocated

Check if CQI is pending for this CC

Check if DL alloc is pending

If UL grant allocated in current carrier

Returns:

UCI_PUSCH_NONE

Uci_alloc

Code:

```
//! Finds eNB CC Idex that currently holds UCI
uci_pusch_t is_uci_included(const sf_sched*        sf_sched,
                            const sf_sched_result& other_cc_results,
                            const sched_ue*        user,
                            uint32_t               current_enb_cc_idx)
{
  uci_pusch_t uci_alloc = UCI_PUSCH_NONE;

  if (not user->get_active_cell_index(current_enb_cc_idx).first) {
    return UCI_PUSCH_NONE;
  }

  // Check if UCI needs to be allocated
  const sched_interface::ue_cfg_t& ue_cfg = user->get_ue_cfg();
  for (uint32_t enbccidx = 0; enbccidx < other_cc_results.enb_cc_list.size() and uci_alloc != UCI_PUSCH_ACK_CQI;
       ++enbccidx) {
    auto p = user->get_active_cell_index(enbccidx);
    if (not p.first) {
      continue;
    }
    uint32_t ueccidx = p.second;

    // Check if CQI is pending for this CC
    const srsran_cqi_report_cfg_t& cqi_report = ue_cfg.supported_cc_list[ueccidx].dl_cfg.cqi_report;
    if (srsran_cqi_periodic_send(&cqi_report, sf_sched->get_tti_tx_ul().to_uint(), SRSRAN_FDD)) {
      if (uci_alloc == UCI_PUSCH_ACK) {
        uci_alloc = UCI_PUSCH_ACK_CQI;
      } else {
        uci_alloc = UCI_PUSCH_CQI;
      }
    }


    // Check if DL alloc is pending
    bool needs_ack_uci = false;
    if (enbccidx == current_enb_cc_idx) {
      needs_ack_uci = sf_sched->is_dl_alloc(user->get_rnti());
    } else {
      auto& dl_result = other_cc_results.enb_cc_list[enbccidx].dl_sched_result;
      for (uint32_t j = 0; j < dl_result.data.size(); ++j) {
        if (dl_result.data[j].dci.rnti == user->get_rnti()) {
          needs_ack_uci = true;
          break;
        }
      }
    }
    if (needs_ack_uci) {
      if (uci_alloc == UCI_PUSCH_CQI) {
        // Once we include ACK and CQI, stop the search
        uci_alloc = UCI_PUSCH_ACK_CQI;
      } else {
        uci_alloc = UCI_PUSCH_ACK;
      }
    }
  }
  if (uci_alloc == UCI_PUSCH_NONE) {
    return uci_alloc;
  }
```

```
  // If UL grant allocated in current carrier
  uint32_t ue_cc_idx      = other_cc_results.enb_cc_list.size();
  int      sel_enb_cc_idx = -1;
  if (sf_sched->is_ul_alloc(user->get_rnti())) {
    ue_cc_idx      = user->get_active_cell_index(current_enb_cc_idx).second;
    sel_enb_cc_idx = current_enb_cc_idx;
  }

  for (uint32_t enbccidx = 0; enbccidx < other_cc_results.enb_cc_list.size(); ++enbccidx) {
    for (uint32_t j = 0; j < other_cc_results.enb_cc_list[enbccidx].ul_sched_result.pusch.size(); ++j) {
      // Checks all the UL grants already allocated for the given rnti
      if (other_cc_results.enb_cc_list[enbccidx].ul_sched_result.pusch[j].dci.rnti == user->get_rnti()) {
        auto p = user->get_active_cell_index(enbccidx);
        // If the UE CC Idx is the lowest so far
        if (p.first and p.second < ue_cc_idx) {
          ue_cc_idx      = p.second;
          sel_enb_cc_idx = enbccidx;
        }
      }
    }
  }
  if (sel_enb_cc_idx == (int)current_enb_cc_idx) {
    return uci_alloc;
  } else {
    return UCI_PUSCH_NONE;
  }
}
```

Method Name:

void sf_sched::set_ul_sched_result()

Parameters:

const sf_cch_allocator::alloc_result_t& dci_result

sched_interface::ul_sched_res_t* ul_result

sched_ue_list& ue_list

Functionality:

Set UL data DCI locs and format

Checks if UCI is encoded in the current carrier

Generate DCI Format 1A

Print Resulting UL Allocation

Returns:

void

Code:

```
void sf_sched::set_ul_sched_result(const sf_cch_allocator::alloc_result_t& dci_result,
                                   sched_interface::ul_sched_res_t*        ul_result,
                                   sched_ue_list&                          ue_list)
{
  /* Set UL data DCI locs and format */
  for (const auto& ul_alloc : ul_data_allocs) {
    auto ue_it = ue_list.find(ul_alloc.rnti);
    if (ue_it == ue_list.end()) {
      continue;
    }
    sched_ue* user = ue_it->second.get();

    srsran_dci_location_t cce_range = {0, 0};
    if (ul_alloc.needs_pdcch()) {
      cce_range = dci_result[ul_alloc.dci_idx]->dci_pos;
    }

    // If UCI is encoded in the current carrier
    uci_pusch_t uci_type = is_uci_included(this, *cc_results, user, cc_cfg->enb_cc_idx);

    /* Generate DCI Format1A */
    ul_result->pusch.emplace_back();
    sched_interface::ul_sched_data_t& pusch = ul_result->pusch.back();
    uint32_t total_data_before              = user->get_pending_ul_data_total(get_tti_tx_ul(), cc_cfg->enb_cc_idx);
    int      tbs                            = user->generate_format0(&pusch,
                                      get_tti_tx_ul(),
                                      cc_cfg->enb_cc_idx,
                                      ul_alloc.alloc,
                                      ul_alloc.needs_pdcch(),
                                      cce_range,
                                      ul_alloc.msg3_mcs,
                                      uci_type);



    ul_harq_proc* h                 = user->get_ul_harq(get_tti_tx_ul(), cc_cfg->enb_cc_idx);
    uint32_t      new_pending_bytes = user->get_pending_ul_new_data(get_tti_tx_ul(), cc_cfg->enb_cc_idx);
    // Allow TBS=0 in case of UCI-only PUSCH
    if (tbs < 0 || (tbs == 0 && pusch.dci.tb.mcs_idx != 29)) {
      fmt::memory_buffer str_buffer;
      fmt::format_to(str_buffer,
                     "SCHED: Error {} {} rnti=0x{:x}, pid={}, dci=({},{}), prb={}, bsr={}",
                     ul_alloc.is_msg3 ? "Msg3" : "UL",
                     ul_alloc.is_retx() ? "retx" : "tx",
                     user->get_rnti(),
                     h->get_id(),
                     pusch.dci.location.L,
                     pusch.dci.location.ncce,
                     ul_alloc.alloc,
                     new_pending_bytes);
      logger.warning("%s", srsran::to_c_str(str_buffer));
      ul_result->pusch.pop_back();
      continue;
    }
```

```
    // Print Resulting UL Allocation
    uint32_t old_pending_bytes = user->get_pending_ul_old_data();
    if (logger.info.enabled()) {
      fmt::memory_buffer str_buffer;
      fmt::format_to(str_buffer,
                     "SCHED: {} {} rnti=0x{:x}, cc={}, pid={}, dci=({},{}), prb={}, n_rtx={}, tbs={}, bsr={} ({}-{})",
                     ul_alloc.is_msg3 ? "Msg3" : "UL",
                     ul_alloc.is_retx() ? "retx" : "tx",
                     user->get_rnti(),
                     cc_cfg->enb_cc_idx,
                     h->get_id(),
                     pusch.dci.location.L,
                     pusch.dci.location.ncce,
                     ul_alloc.alloc,
                     h->nof_retx(0),
                     tbs,
                     new_pending_bytes,
                     total_data_before,
                     old_pending_bytes);
      logger.info("%s", srsran::to_c_str(str_buffer));
    }

    pusch.current_tx_nb = h->nof_retx(0);
  }
}
```

Method Name:

alloc_result sf_sched::alloc_msg3()

Parameters:

sched_ue* user

const sched_interface::dl_sched_rar_grant_t& rargrant

Functionality:

Derive PRBs from allocated RAR grants

Sets allocation results ret

Allocates msg3s

Returns:

ret

Code:

```
alloc_result sf_sched::alloc_msg3(sched_ue* user, const sched_interface::dl_sched_rar_grant_t& rargrant)
{
  // Derive PRBs from allocated RAR grants
  prb_interval msg3_alloc = prb_interval::riv_to_prbs(rargrant.grant.rba, cc_cfg->nof_prb());

  alloc_result ret = alloc_ul(user, msg3_alloc, sf_sched::ul_alloc_t::NEWTX, true, rargrant.grant.trunc_mcs);
  if (ret != alloc_result::success) {
    fmt::memory_buffer str_buffer;
    fmt::format_to(str_buffer, "{}", msg3_alloc);
    logger.warning("SCHED: Could not allocate msg3 within %s.", srsran::to_c_str(str_buffer));
  }
  return ret;
}
```

Method Name:

void sf_sched::generate_sched_results()

Parameters:

sched_ue_list& ue_db

Functionality:

Resume UL HARQs with pending retxs that did not get allocated

Picks one of the possible DCI masks

Register final CFI

Generate DCI formats and fill sched_result structs

Stores remaining sf_sched results for this TTI

Returns:

void

Code:

```
void sf_sched::generate_sched_results(sched_ue_list& ue_db)
{
  cc_sched_result* cc_result = cc_results->get_cc(cc_cfg->enb_cc_idx);

  /* Resume UL HARQs with pending retxs that did not get allocated */
  using phich_t    = sched_interface::ul_sched_phich_t;
  auto& phich_list = cc_result->ul_sched_result.phich;
  for (uint32_t i = 0; i < cc_result->ul_sched_result.phich.size(); ++i) {
    auto& phich = phich_list[i];
    if (phich.phich == phich_t::NACK) {
      auto&         ue = *ue_db[phich.rnti];
      ul_harq_proc* h  = ue.get_ul_harq(get_tti_tx_ul(), cc_cfg->enb_cc_idx);
      if (not is_ul_alloc(ue.get_rnti()) and h != nullptr and not h->is_empty()) {
        // There was a missed UL harq retx. Halt+Resume the HARQ
        phich.phich = phich_t::ACK;
        logger.debug("SCHED: rnti=0x%x UL harq pid=%d is being resumed", ue.get_rnti(), h->get_id());
      }
    }
  }
```

```
  /* Pick one of the possible DCI masks */
  sf_cch_allocator::alloc_result_t dci_result;
  //  tti_alloc.get_pdcch_grid().result_to_string();
  tti_alloc.get_pdcch_grid().get_allocs(&dci_result, &cc_result->pdcch_mask);

  /* Register final CFI */
  cc_result->dl_sched_result.cfi = tti_alloc.get_pdcch_grid().get_cfi();

  /* Generate DCI formats and fill sched_result structs */
  for (const auto& bc_alloc : bc_allocs) {
    cc_result->dl_sched_result.bc.emplace_back(bc_alloc.bc_grant);
    cc_result->dl_sched_result.bc.back().dci.location = dci_result[bc_alloc.dci_idx]->dci_pos;
    log_broadcast_allocation(cc_result->dl_sched_result.bc.back(), bc_alloc.rbg_range, *cc_cfg);
  }

  for (const auto& rar_alloc : rar_allocs) {
    cc_result->dl_sched_result.rar.emplace_back(rar_alloc.rar_grant);
    cc_result->dl_sched_result.rar.back().dci.location = dci_result[rar_alloc.alloc_data.dci_idx]->dci_pos;
    log_rar_allocation(cc_result->dl_sched_result.rar.back(), rar_alloc.alloc_data.rbg_range);
  }

  set_dl_data_sched_result(dci_result, &cc_result->dl_sched_result, ue_db);

  set_ul_sched_result(dci_result, &cc_result->ul_sched_result, ue_db);

  /* Store remaining sf_sched results for this TTI */
  cc_result->dl_mask   = tti_alloc.get_dl_mask();
  cc_result->ul_mask   = tti_alloc.get_ul_mask();
  cc_result->generated = true;
}
```

Method Name:

uint32_t sf_sched::get_nof_ctrl_symbols() const

Parameters:

none

Functionality:

Gets CFI from number of prbs

Returns:

tti_alloc.get_cfi() + ((cc_cfg->cfg.cell.nof_prb <= 10) ? 1 : 0)

Code:

```
uint32_t sf_sched::get_nof_ctrl_symbols() const
{
  return tti_alloc.get_cfi() + ((cc_cfg->cfg.cell.nof_prb <= 10) ? 1 : 0);
}
```

sched_helpers.cc

Method Name:

static srslog::basic_logger& get_mac_logger()

Parameters:

none

Functionality:

Sets MAC logger

Returns:

mac_logger

Code:

```
static srslog::basic_logger& get_mac_logger()
{
  static srslog::basic_logger& mac_logger = srslog::fetch_basic_logger("MAC");
  return mac_logger;
}
```

Method Name:

const char* to_string_short(s)

Parameters:

rsran_dci_format_t dcifmt

Functionality:

Sets all DCI formats

Returns:

0

1

1A

1B

2

2A

2B

unknown

Code:

```
const char* to_string_short(srsran_dci_format_t dcifmt)
{
  switch (dcifmt) {
    case SRSRAN_DCI_FORMAT0:
      return "0";
    case SRSRAN_DCI_FORMAT1:
      return "1";
    case SRSRAN_DCI_FORMAT1A:
      return "1A";
    case SRSRAN_DCI_FORMAT1B:
      return "1B";
    case SRSRAN_DCI_FORMAT2:
      return "2";
    case SRSRAN_DCI_FORMAT2A:
      return "2A";
    case SRSRAN_DCI_FORMAT2B:
      return "2B";
    default:
      return "unknown";
  }
}
```

Method Name:

void fill_dl_cc_result_info()

Parameters:

custom_mem_buffer& strbuf

const dl_sched_data_t& data

Functionality:

Sets first CE

Sets up CC result information for transfer

Returns:

void

Code:

```
void fill_dl_cc_result_info(custom_mem_buffer& strbuf, const dl_sched_data_t& data)
{
  uint32_t first_ce = sched_interface::MAX_RLC_PDU_LIST;
  for (uint32_t i = 0; i < data.nof_pdu_elems[0]; ++i) {
    if (srsran::is_mac_ce(static_cast<srsran::dl_sch_lcid>(data.pdu[i]->lcid))) {
      first_ce = i;
      break;
    }
  }
  if (first_ce == sched_interface::MAX_RLC_PDU_LIST) {
    return;
  }
  const char* prefix = strbuf.size() > 0 ? " | " : "";
  fmt::format_to(strbuf, "{}rnti=0x{:0x}: [", prefix, data.dci.rnti);
  bool ces_found = false;
  for (uint32_t i = 0; i < data.nof_pdu_elems[0]; ++i) {
    const auto& pdu          = data.pdu[0][i];
    prefix                   = (ces_found) ? " | " : "";
    srsran::dl_sch_lcid lcid = static_cast<srsran::dl_sch_lcid>(pdu.lcid);
    if (srsran::is_mac_ce(lcid)) {
      fmt::format_to(strbuf, "{}CE \"{}\"", prefix, srsran::to_string_short(lcid));
      ces_found = true;
    }
  }
  fmt::format_to(strbuf, "]");
}
```

Method Name:

void fill_dl_cc_result_debug()

Parameters:

custom_mem_buffer& strbuf

const dl_sched_data_t& data

Functionality:

Checks number of PDU elements

Sets string format for DCI

Returns:

void

Code:

```
void fill_dl_cc_result_debug(custom_mem_buffer& strbuf, const dl_sched_data_t& data)
{
  if (data.nof_pdu_elems[0] == 0 and data.nof_pdu_elems[1] == 0) {
    return;
  }
  fmt::format_to(strbuf,
                 "  > rnti=0x{:0x}, tbs={}, f={}, mcs={}: [",
                 data.dci.rnti,
                 data.tbs[0],
                 to_string_short(data.dci.format),
                 data.dci.tb[0].mcs_idx);
  for (uint32_t tb = 0; tb < SRSRAN_MAX_TB; ++tb) {
    for (uint32_t i = 0; i < data.nof_pdu_elems[tb]; ++i) {
      const auto&         pdu    = data.pdu[tb][i];
      const char*         prefix = (i == 0) ? "" : " | ";
      srsran::dl_sch_lcid lcid   = static_cast<srsran::dl_sch_lcid>(pdu.lcid);
      if (srsran::is_mac_ce(lcid)) {
        fmt::format_to(strbuf, "{}CE \"{}\"", prefix, srsran::to_string_short(lcid));
      } else {
        fmt::format_to(strbuf, "{}SDU lcid={}, tb={}, len={} B", prefix, pdu.lcid, tb, pdu.nbytes);
      }
    }
  }
  fmt::format_to(strbuf, "]");
}
```

Method Name:

void log_dl_cc_results()

Parameters:

srslog::basic_logger& logger

uint32_t enb_cc_idx

const sched_interface::dl_sched_res_t& result

Functionality:

Checks for logger enabled

Buffers logger information

Buffers logger debug

Logs PDU payload

Logs MAC CEs

Returns:

void

Code:

```
void log_dl_cc_results(srslog::basic_logger& logger, uint32_t enb_cc_idx, const sched_interface::dl_sched_res_t& result)
{
  if (!logger.info.enabled() && !logger.debug.enabled()) {
    return;
  }

  custom_mem_buffer strbuf;
  for (const auto& data : result.data) {
    if (logger.debug.enabled()) {
      fill_dl_cc_result_debug(strbuf, data);
    } else {
      fill_dl_cc_result_info(strbuf, data);
    }
  }
  if (strbuf.size() != 0) {
    if (logger.debug.enabled()) {
      logger.debug("SCHED: DL MAC PDU payload cc=%d:\n%s", enb_cc_idx, srsran::to_c_str(strbuf));
    } else {
      logger.info("SCHED: DL MAC CEs cc=%d: %s", enb_cc_idx, srsran::to_c_str(strbuf));
    }
  }
}
```

Method Name:

void log_phich_cc_results()

Parameters:

srslog::basic_logger& logger

uint32_t enb_cc_idx,

const sched_interface::ul_sched_res_t& result

Functionality:

Sets up PHICH

Sets up PHICH allocation information

Returns:

void

Code:

```
void log_phich_cc_results(srslog::basic_logger&                  logger,
                          uint32_t                               enb_cc_idx,
                          const sched_interface::ul_sched_res_t& result)
{
  using phich_t = sched_interface::ul_sched_phich_t;
  if (!logger.debug.enabled()) {
    return;
  }
  custom_mem_buffer strbuf;
  for (uint32_t i = 0; i < result.phich.size(); ++i) {
    const phich_t& phich  = result.phich[i];
    const char*    prefix = strbuf.size() > 0 ? " | " : "";
    const char*    val    = phich.phich == phich_t::ACK ? "ACK" : "NACK";
    fmt::format_to(strbuf, "{}rnti=0x{:0x}, val={}", prefix, phich.rnti, val);
  }
  if (strbuf.size() != 0) {
    logger.debug("SCHED: Allocated PHICHs, cc=%d: [%s]", enb_cc_idx, srsran::to_c_str(strbuf));
  }
}
```

Method Name:

prb_interval prb_interval::rbgs_to_prbs()

Parameters:

cconst rbg_interval& rbgs

uint32_t cell_nof_prb

Functionality:

Sets PRB interval

Returns:

prb_interval{rbgs.start() * P, std::min(rbgs.stop() * P, cell_nof_prb)}

Code:

```
prb_interval prb_interval::rbgs_to_prbs(const rbg_interval& rbgs, uint32_t cell_nof_prb)
{
  uint32_t P = srsran_ra_type0_P(cell_nof_prb);
  return prb_interval{rbgs.start() * P, std::min(rbgs.stop() * P, cell_nof_prb)};
}
```

Method Name:

rbg_interval rbg_interval::rbgmask_to_rbgs()

Parameters:

const rbgmask_t& mask

Functionality:

Builds rbg interval

Returns:

rbg_interval

Code:

```
rbg_interval rbg_interval::rbgmask_to_rbgs(const rbgmask_t& mask)
{
  int rb_start = -1;
  for (uint32_t i = 0; i < mask.size(); i++) {
    if (rb_start == -1) {
      if (mask.test(i)) {
        rb_start = i;
      }
    } else {
      if (!mask.test(i)) {
        return rbg_interval(rb_start, i);
      }
    }
  }
  if (rb_start != -1) {
    return rbg_interval(rb_start, mask.size());
  } else {
    return rbg_interval();
  }
}
```

Method Name:

prb_interval prb_interval::riv_to_prbs()

Parameters:

uint32_t riv

uint32_t nof_prbs

int nof_vrbs

Functionality:

Sets number of VRBs

Returns:

{rb_start, rb_start + l_crb}

Code:

```
prb_interval prb_interval::riv_to_prbs(uint32_t riv, uint32_t nof_prbs, int nof_vrbs)
{
  if (nof_vrbs < 0) {
    nof_vrbs = nof_prbs;
  }
  uint32_t rb_start, l_crb;
  srsran_ra_type2_from_riv(riv, &l_crb, &rb_start, nof_prbs, (uint32_t)nof_vrbs);
  return {rb_start, rb_start + l_crb};
}
```

Method Name:

bool is_contiguous()

Parameters:

const rbgmask_t& mask

Functionality:

Checks if RBG is contiguous

Returns:

True

False

Code:

```
bool is_contiguous(const rbgmask_t& mask)
{
  return rbg_interval::rbgmask_to_rbgs(mask).length() == mask.count();
}
```

Method Name:

sched_cell_params_t::dl_nof_re_table generate_nof_re_table()

Parameters:

const srsran_cell_t& cell

Functionality:

Generates number of RE tables

Builds all RE scheduler tables

Returns:

table

Code:

```
sched_cell_params_t::dl_nof_re_table generate_nof_re_table(const srsran_cell_t& cell)
{
  sched_cell_params_t::dl_nof_re_table table(cell.nof_prb);

  srsran_dl_sf_cfg_t dl_sf    = {};
  dl_sf.sf_type               = SRSRAN_SF_NORM;
  dl_sf.tdd_config.configured = false;

  for (uint32_t cfi = 0; cfi < SRSRAN_NOF_CFI; ++cfi) {
    dl_sf.cfi = cfi + 1;
    for (uint32_t sf_idx = 0; sf_idx < SRSRAN_NOF_SF_X_FRAME; ++sf_idx) {
      dl_sf.tti = sf_idx;
      for (uint32_t s = 0; s < SRSRAN_NOF_SLOTS_PER_SF; ++s) {
        for (uint32_t n = 0; n < cell.nof_prb; ++n) {
          table[n][sf_idx][s][cfi] = ra_re_x_prb(&cell, &dl_sf, s, n);
        }
      }
    }
  }
  return table;
}
```

Method Name:

sched_cell_params_t::dl_lb_nof_re_table get_lb_nof_re_x_prb()

Parameters:

const sched_cell_params_t::dl_nof_re_table& table

Functionality:

Adjusts ret for table size

Sets PRB RE vector

Transforms vector from table size

Returns:

ret

Code:

```
sched_cell_params_t::dl_lb_nof_re_table get_lb_nof_re_x_prb(const sched_cell_params_t::dl_nof_re_table& table)
{
  sched_cell_params_t::dl_lb_nof_re_table ret;
  for (uint32_t sf_idx = 0; sf_idx < SRSRAN_NOF_SF_X_FRAME; ++sf_idx) {
    ret[sf_idx].resize(table.size());
    srsran::bounded_vector<uint32_t, SRSRAN_MAX_PRB> re_prb_vec(table.size());
    for (uint32_t p = 0; p < table.size(); ++p) {
      for (uint32_t s = 0; s < SRSRAN_NOF_SLOTS_PER_SF; ++s) {
        // assume max CFI to compute lower bound
        re_prb_vec[p] += table[p][sf_idx][s][SRSRAN_NOF_CFI - 1];
      }
    }

    srsran::bounded_vector<uint32_t, SRSRAN_MAX_PRB> re_prb_vec2(re_prb_vec.size());
    std::copy(re_prb_vec.begin(), re_prb_vec.end(), re_prb_vec2.begin());
    // pick intervals of PRBs with the lowest sum of REs
    ret[sf_idx][0] = *std::min_element(re_prb_vec2.begin(), re_prb_vec2.end());
    for (uint32_t p = 1; p < table.size(); ++p) {
      std::transform(re_prb_vec2.begin(),
                     re_prb_vec2.end() - 1,
                     re_prb_vec.begin() + p,
                     re_prb_vec2.begin(),
                     std::plus<uint32_t>());
      re_prb_vec2.pop_back();
      ret[sf_idx][p] = *std::min_element(re_prb_vec2.begin(), re_prb_vec2.end());
    }
  }
  return ret;
}
```

Method Name:

void sched_cell_params_t::regs_deleter::operator()()

Parameters:

srsran_regs_t* p

Functionality:

Deletes all srsRAN regulations

Returns:

void

Code:

```
void sched_cell_params_t::regs_deleter::operator()(srsran_regs_t* p)
{
  if (p != nullptr) {
    srsran_regs_free(p);
    delete p;
  }
}
```

Method Name:

bool sched_cell_params_t::set_cfg()

Parameters:

uint32_t enb_cc_idx_

const sched_interface::cell_cfg_t& cfg_

const sched_interface::sched_args_t& sched_args

Functionality:

Checks basic cell configuration

Checks if PUSCH has space for msg3

Checks if PRACH fits in PUSCH + PUCCH space

Sets scheduler parameters

Initializes regs

Computes Common locations for DCI for each CFI

Computes UE locations for RA-RNTI

precompute number of cces in PDCCH for each CFI

Configures PUCCH struct for position derivation

Returns:

True

False

Code:

```
bool invalid_prach;
if (cfg.cell.nof_prb == 6) {
  // PUCCH has to allow space for Msg3
  if (cfg.nrb_pucch > 1) {
    srsran::console("Invalid PUCCH configuration: nrb_pucch=%d does not allow space for Msg3 transmission..\n",
                    cfg.nrb_pucch);
    return false;
  }
  // PRACH has to fit within the PUSCH+PUCCH space
  invalid_prach = cfg.prach_freq_offset + 6 > cfg.cell.nof_prb;
} else {
  // PRACH has to fit within the PUSCH space
  invalid_prach = (cfg.prach_freq_offset + 6) > (cfg.cell.nof_prb - cfg.nrb_pucch) or
                  ((int)cfg.prach_freq_offset < cfg.nrb_pucch);
}
if (invalid_prach) {
  Error("Invalid PRACH configuration: frequency offset=%d outside bandwidth limits", cfg.prach_freq_offset);
  srsran::console("Invalid PRACH configuration: frequency offset=%d outside bandwidth limits\n",
                  cfg.prach_freq_offset);
  return false;
}

// Set derived sched parameters

// init regs
regs.reset(new srsran_regs_t{});
if (srsran_regs_init(regs.get(), cfg.cell) != SRSRAN_SUCCESS) {
  Error("Getting DCI locations");
  return false;
}
```

```
  // Compute Common locations for DCI for each CFI
  for (uint32_t cfix = 0; cfix < SRSRAN_NOF_CFI; cfix++) {
    generate_cce_location(regs.get(), common_locations[cfix], cfix + 1);
  }
  if (common_locations[sched_cfg->max_nof_ctrl_symbols - 1][2].empty()) {
    Error("SCHED: Current cfi=%d is not valid for broadcast (check scheduler.max_nof_ctrl_symbols in conf file).",
          sched_cfg->max_nof_ctrl_symbols);
    srsran::console(
        "SCHED: Current cfi=%d is not valid for broadcast (check scheduler.max_nof_ctrl_symbols in conf file).\n",
        sched_cfg->max_nof_ctrl_symbols);
    return false;
  }

  // Compute UE locations for RA-RNTI
  for (uint32_t cfi = 0; cfi < SRSRAN_NOF_CFI; cfi++) {
    for (uint32_t sf_idx = 0; sf_idx < SRSRAN_NOF_SF_X_FRAME; sf_idx++) {
      generate_cce_location(regs.get(), rar_locations[sf_idx][cfi], cfi + 1, sf_idx);
    }
  }

  // precompute nof cces in PDCCH for each CFI
  for (uint32_t cfix = 0; cfix < nof_cce_table.size(); ++cfix) {
    int ret = srsran_regs_pdcch_ncce(regs.get(), cfix + 1);
    if (ret < 0) {
      Error("SCHED: Failed to calculate the number of CCEs in the PDCCH");
      return false;
    }
    nof_cce_table[cfix] = (uint32_t)ret;
  }



  // PUCCH config struct for PUCCH position derivation
  pucch_cfg_common.format            = SRSRAN_PUCCH_FORMAT_1;
  pucch_cfg_common.delta_pucch_shift = cfg.delta_pucch_shift;
  pucch_cfg_common.n_rb_2            = cfg.nrb_cqi;
  pucch_cfg_common.N_cs              = cfg.ncs_an;
  pucch_cfg_common.N_pucch_1         = cfg.n1pucch_an;

  P        = srsran_ra_type0_P(cfg.cell.nof_prb);
  nof_rbgs = srsran::ceil_div(cfg.cell.nof_prb, P);

  nof_re_table    = generate_nof_re_table(cfg.cell);
  nof_re_lb_table = get_lb_nof_re_x_prb(nof_re_table);

  return true;
}
```

Method Name:

uint32_t sched_cell_params_t::get_dl_lb_nof_re() const

Parameters:

tti_point tti_tx_dl

uint32_t nof_prbs_alloc

Functionality:

Ensures that allocated PRBs does not exceed number of PRBs

Gets number of REs

Returns:

0

nof_re

Code:

```
uint32_t sched_cell_params_t::get_dl_lb_nof_re(tti_point tti_tx_dl, uint32_t nof_prbs_alloc) const
{
  assert(nof_prbs_alloc <= nof_prb());
  if (nof_prbs_alloc == 0) {
    return 0;
  }
  uint32_t sf_idx = tti_tx_dl.sf_idx();
  uint32_t nof_re = nof_re_lb_table[sf_idx][nof_prbs_alloc - 1];

  // sanity check
  assert(nof_re <= srsran_ra_dl_approx_nof_re(&cfg.cell, nof_prbs_alloc, SRSRAN_NOF_CFI));
  return nof_re;
}
```

Method Name:

uint32_t sched_cell_params_t::get_dl_nof_res() const

Parameters:

srsran::tti_point tti_tx_dl

const srsran_dci_dl_t& dci

uint32_t cfi

Functionality:

Ensures CFI is correct

Checks number of REs from RE table

Returns:

nof_re

Code:

```
uint32_t
sched_cell_params_t::get_dl_nof_res(srsran::tti_point tti_tx_dl, const srsran_dci_dl_t& dci, uint32_t cfi) const
{
  assert(cfi > 0 && "CFI has to be within (1..3)");
  srsran_pdsch_grant_t grant = {};
  srsran_dl_sf_cfg_t   dl_sf = {};
  dl_sf.cfi                  = cfi;
  dl_sf.tti                  = tti_tx_dl.to_uint();
  srsran_ra_dl_grant_to_grant_prb_allocation(&dci, &grant, nof_prb());

  uint32_t nof_re = 0;
  for (uint32_t p = 0; p < nof_prb(); ++p) {
    for (uint32_t s = 0; s < SRSRAN_NOF_SLOTS_PER_SF; ++s) {
      if (grant.prb_idx[s][p]) {
        nof_re += nof_re_table[p][tti_tx_dl.sf_idx()][s][cfi - 1];
      }
    }
  }

  return nof_re;
}
```

Method Name:

cce_frame_position_table generate_cce_location_table()

Parameters:

uint16_t rnti

const sched_cell_params_t& cell_cfg

Functionality:

Generate allowed CCE locations

Returns:

dci_locations

Code:

```
cce_frame_position_table generate_cce_location_table(uint16_t rnti, const sched_cell_params_t& cell_cfg)
{
  cce_frame_position_table dci_locations = {};
  // Generate allowed CCE locations
  for (int cfi = 0; cfi < SRSRAN_NOF_CFI; cfi++) {
    for (int sf_idx = 0; sf_idx < SRSRAN_NOF_SF_X_FRAME; sf_idx++) {
      generate_cce_location(cell_cfg.regs.get(), dci_locations[sf_idx][cfi], cfi + 1, sf_idx, rnti);
    }
  }
  return dci_locations;
}
```

Method Name:

void generate_cce_location()

Parameters:

srsran_regs_t* regs_

cce_cfi_position_table& locations

uint32_t cfi

uint32_t sf_idx

uint16_t rnti

Functionality:

Gets DCI location

Saves location in a different format

Returns:

void

Code:

```
void generate_cce_location(srsran_regs_t*          regs_,
                           cce_cfi_position_table& locations,
                           uint32_t                cfi,
                           uint32_t                sf_idx,
                           uint16_t                rnti)
{
  locations = {};

  srsran_dci_location_t loc[64];
  uint32_t              nloc = 0;
  if (rnti == 0) {
    nloc = srsran_pdcch_common_locations_ncce(srsran_regs_pdcch_ncce(regs_, cfi), loc, 64);
  } else {
    nloc = srsran_pdcch_ue_locations_ncce(srsran_regs_pdcch_ncce(regs_, cfi), loc, 64, sf_idx, rnti);
  }

  // Save to different format
  for (uint32_t i = 0; i < nloc; i++) {
    uint32_t l = loc[i].L;
    locations[l].push_back(loc[i].ncce);
  }
}
```

Method Name:

uint32_t get_aggr_level()

Parameters:

uint32_t nof_bits

uint32_t dl_cqi

uint32_t max_aggr_lvl

uint32_t cell_nof_prb

bool use_tbs_index_alt

Functionality:

Sets max level to min of l_max and l_aggr_max

Returns:

l

Code:

```
uint32_t
get_aggr_level(uint32_t nof_bits, uint32_t dl_cqi, uint32_t max_aggr_lvl, uint32_t cell_nof_prb, bool use_tbs_index_alt)
{
  uint32_t l            = 0;
  float    max_coderate = srsran_cqi_to_coderate(dl_cqi, use_tbs_index_alt);
  float    coderate;
  float    factor = 1.5;
  uint32_t l_max  = 3;
  if (cell_nof_prb == 6) {
    factor = 1.0;
    l_max  = 2;
  }
  l_max = SRSRAN_MIN(max_aggr_lvl, l_max);

  do {
    coderate = srsran_pdcch_coderate(nof_bits, l);
    l++;
  } while (l < l_max && factor * coderate > max_coderate);

  Debug("SCHED: CQI=%d, l=%d, nof_bits=%d, coderate=%.2f, max_coderate=%.2f",
        dl_cqi,
        l,
        nof_bits,
        coderate,
        max_coderate);
  return l;
}
```

Method Name:

int check_ue_cfg_correctness()

Parameters:

const sched_interface::ue_cfg_t& ue_cfg

Functionality:

In case of CA, CQI configs must exist and cannot collide in the PUCCH

Sets ret based on CC list size

Checks UE and CC configuration

Returns:

ret

Code:

```
/// sanity check the UE CC configuration
int check_ue_cfg_correctness(const sched_interface::ue_cfg_t& ue_cfg)
{
  using cc_t             = sched_interface::ue_cfg_t::cc_cfg_t;
  const auto& cc_list    = ue_cfg.supported_cc_list;
  bool        has_scells = std::count_if(cc_list.begin(), cc_list.end(), [](const cc_t& c) { return c.active; }) > 1;
  int         ret        = SRSRAN_SUCCESS;

  if (has_scells) {
    // In case of CA, CQI configs must exist and cannot collide in the PUCCH
    for (uint32_t i = 0; i < cc_list.size(); ++i) {
      const auto& cc1 = cc_list[i];
      if (not cc1.active) {
        continue;
      }
      if (not cc1.dl_cfg.cqi_report.periodic_configured and not cc1.dl_cfg.cqi_report.aperiodic_configured) {
        Warning("SCHED: No CQI configuration was provided for UE scell index=%d", i);
        ret = SRSRAN_ERROR;
      } else if (cc1.dl_cfg.cqi_report.periodic_configured) {
        for (uint32_t j = i + 1; j < cc_list.size(); ++j) {
          if (cc_list[j].active and cc_list[j].dl_cfg.cqi_report.periodic_configured and
              cc_list[j].dl_cfg.cqi_report.pmi_idx == cc1.dl_cfg.cqi_report.pmi_idx) {
            Warning("SCHED: The provided CQI configurations for UE scells %d and %d collide in time resources.", i, j);
            ret = SRSRAN_ERROR;
          }
        }
      }
    }
  }
  return ret;
}
```

Method Name:

const char* to_string()

Parameters:

sched_interface::ue_bearer_cfg_t::direction_t dir

Functionality:

Sets UE bearer configuration messages

Returns:

Idle

Bi-dir

DL

UL

unrecognized direction

Code:

```
const char* to_string(sched_interface::ue_bearer_cfg_t::direction_t dir)
{
  switch (dir) {
    case sched_interface::ue_bearer_cfg_t::IDLE:
      return "idle";
    case sched_interface::ue_bearer_cfg_t::BOTH:
      return "bi-dir";
    case sched_interface::ue_bearer_cfg_t::DL:
      return "DL";
    case sched_interface::ue_bearer_cfg_t::UL:
      return "UL";
    default:
      return "unrecognized direction";
  }
}
```

sched_ue.cc

Method Name:

sched_ue::sched_ue():logger(srslog::fetch_basic_logger("MAC"))

Parameters:

uint16_t rnti_

const std::vector<sched_cell_params_t>& cell_list_params_

const ue_cfg_t& cfg_

Functionality:

Sets up rnti and cell configurations

Returns:

none

Code:

```
sched_ue::sched_ue(uint16_t rnti_, const std::vector<sched_cell_params_t>& cell_list_params_, const ue_cfg_t& cfg_) :
  logger(srslog::fetch_basic_logger("MAC"))
{
  rnti = rnti_;
  cells.reserve(cell_list_params_.size());
  for (auto& c : cell_list_params_) {
    cells.emplace_back(rnti_, c, current_tti);
  }
  logger.info("SCHED: Added user rnti=0x%x", rnti);

  set_cfg(cfg_);
}
```

Method Name:

void sched_ue::set_cfg()

Parameters:

const ue_cfg_t& cfg_

Functionality:

for the first configured cc, set it as primary cc

updates configuration

updates bearer cfgs

updates ue cells

Checks for correct UE configuration

Returns:

void

Code:

```
void sched_ue::set_cfg(const ue_cfg_t& cfg_)
{
  // for the first configured cc, set it as primary cc
  if (cfg.supported_cc_list.empty()) {
    uint32_t primary_cc_idx = 0;
    if (not cfg_.supported_cc_list.empty()) {
      primary_cc_idx = cfg_.supported_cc_list[0].enb_cc_idx;
    } else {
      logger.warning("Primary cc idx not provided in scheduler ue_cfg. Defaulting to cc_idx=0");
    }
    // setup primary cc
    main_cc_params = cells[primary_cc_idx].cell_cfg;
    cell           = main_cc_params->cfg.cell;
    max_msg3retx   = main_cc_params->cfg.maxharq_msg3tx;
  }

  // update configuration
  std::vector<sched::ue_cfg_t::cc_cfg_t> prev_supported_cc_list = std::move(cfg.supported_cc_list);
  cfg                                                           = cfg_;

  // update bearer cfgs
  lch_handler.set_cfg(cfg_);


 // update ue cells
 bool scell_activation_state_changed = false;
 for (auto& c : cells) {
   c.set_ue_cfg(cfg);
   scell_activation_state_changed |=
       c.is_scell() and (c.cc_state() == cc_st::activating or c.cc_state() == cc_st::deactivating);
 }
 bool is_handover = not prev_supported_cc_list.empty() and
                    prev_supported_cc_list[0].enb_cc_idx != cfg.supported_cc_list[0].enb_cc_idx;
 if (prev_supported_cc_list.empty() or is_handover) {
   logger.info("SCHED: rnti=0x%x PCell is now enb_cc_idx=%d", rnti, cfg.supported_cc_list[0].enb_cc_idx);
 }
 if (scell_activation_state_changed and not is_handover) {
   lch_handler.pending_ces.emplace_back(srsran::dl_sch_lcid::SCELL_ACTIVATION);
   logger.info("SCHED: Enqueueing SCell Activation CMD for rnti=0x%x", rnti);
 }

 check_ue_cfg_correctness(cfg);
}
```

Method Name:

void sched_ue::new_subframe()

Parameters:

tti_point tti_rx

uint32_t enb_cc_idx

Functionality:

Sets current tti to tti receive

Checks for cc cells configuration

Returns:

void

Code:

```
void sched_ue::new_subframe(tti_point tti_rx, uint32_t enb_cc_idx)
{
  if (current_tti != tti_rx) {
    current_tti = tti_rx;
    lch_handler.new_tti();
    for (auto& cc : cells) {
      if (cc.configured()) {
        cc.harq_ent.new_tti(tti_rx);
      }
    }
  }

  if (cells[enb_cc_idx].configured()) {
    cells[enb_cc_idx].tpc_fsm.new_tti();
  }
}
```

Method Name:

void sched_ue::set_bearer_cfg()

Parameters:

uint32_t lc_id

const bearer_cfg_t& cfg_

Functionality:

Configures UE bearers

Configures LC id

Returns:

void

Code:

```
void sched_ue::set_bearer_cfg(uint32_t lc_id, const bearer_cfg_t& cfg_)
{
  cfg.ue_bearers[lc_id] = cfg_;
  lch_handler.config_lcid(lc_id, cfg_);
}
```

Method Name:

void sched_ue::rem_bearer()

Parameters:

uint32_t lc_id

Functionality:

Configures UE bearers from LC id

Returns:

void

Code:

```
void sched_ue::rem_bearer(uint32_t lc_id)
{
  cfg.ue_bearers[lc_id] = sched_interface::ue_bearer_cfg_t{};
  lch_handler.config_lcid(lc_id, sched_interface::ue_bearer_cfg_t{});
}
```

Method Name:

void sched_ue::phy_config_enabled()

Parameters:

tti_point tti_rx

bool enabled

Functionality:

Enables physical layer configuration

Returns:

void

Code:

```
void sched_ue::phy_config_enabled(tti_point tti_rx, bool enabled)
{
  for (sched_ue_cell& c : cells) {
    if (c.configured()) {
      c.dl_cqi_tti_rx = tti_rx;
    }
  }
  phy_config_dedicated_enabled = enabled;
}
```

Method Name:

void sched_ue::ul_buffer_state()

Parameters:

uint8_t lcg_id

uint32_t bsr

Functionality:

Set lch handler for upload from buffer state

Returns:

void

Code:

```
void sched_ue::ul_buffer_state(uint8_t lcg_id, uint32_t bsr)
{
  lch_handler.ul_bsr(lcg_id, bsr);
}
```

Method Name:

void sched_ue::ul_buffer_add()

Parameters:

uint8_t lcid

uint32_t bytes

Functionality:

Adds buffer for lch handler for upload

Returns:

void

Code:

```
void sched_ue::ul_buffer_add(uint8_t lcid, uint32_t bytes)
{
  lch_handler.ul_buffer_add(lcid, bytes);
}
```

Method Name:

void sched_ue::ul_phr()

Parameters:

int phr

Functionality:

Sets phr for configured cells

Returns:

void

Code:

```
void sched_ue::ul_phr(int phr)
{
  cells[cfg.supported_cc_list[0].enb_cc_idx].tpc_fsm.set_phr(phr);
}
```

Method Name:

void sched_ue::dl_buffer_state()

Parameters:

uint8_t lc_id

uint32_t tx_queue

uint32_t retx_queue

Functionality:

Sets download buffer state

Returns:

void

Code:

```
void sched_ue::dl_buffer_state(uint8_t lc_id, uint32_t tx_queue, uint32_t retx_queue)
{
  lch_handler.dl_buffer_state(lc_id, tx_queue, retx_queue);
}
```

Method Name:

void sched_ue::mac_buffer_state()

Parameters:

uint32_t ce_code

uint32_t nof_cmds

Functionality:

Sets mac layer buffer state

Returns:

void

Code:

```
void sched_ue::mac_buffer_state(uint32_t ce_code, uint32_t nof_cmds)
{
  auto cmd = (lch_ue_manager::ce_cmd)ce_code;
  for (uint32_t i = 0; i < nof_cmds; ++i) {
    if (cmd == lch_ue_manager::ce_cmd::CON_RES_ID) {
      lch_handler.pending_ces.push_front(cmd);
    } else {
      lch_handler.pending_ces.push_back(cmd);
    }
  }
  logger.info("SCHED: %s for rnti=0x%x needs to be scheduled", to_string(cmd), rnti);
}
```

Method Name:

void sched_ue::set_sr()

Parameters:

none

Functionality:

Sets SR

Returns:

void

Code:

```
void sched_ue::set_sr()
{
  sr = true;
}
```

Method Name:

void sched_ue::unset_sr()

Parameters:

none

Functionality:

Unsets SR

Returns:

void

Code:

```
void sched_ue::unset_sr()
{
  sr = false;
}
```

Method Name:

tti_point prev_meas_gap_start()

Parameters:

tti_point tti

uint32_t period

uint32_t offset

Functionality:

Gets tti point

Returns:

tti_point{static_cast<uint32_t>(floor(static_cast<float>((tti - offset).to_uint()) / period)) * period + offset}

Code:

```
tti_point prev_meas_gap_start(tti_point tti, uint32_t period, uint32_t offset)
{
  return tti_point{static_cast<uint32_t>(floor(static_cast<float>((tti - offset).to_uint()) / period)) * period +
                   offset};
}
```

Method Name:

tti_point next_meas_gap_start()

Parameters:

tti_point tti

uint32_t period

uint32_t offset

Functionality:

Gets previous measured gap

Returns:

prev_meas_gap_start(tti, period, offset) + period

Code:

```
tti_point next_meas_gap_start(tti_point tti, uint32_t period, uint32_t offset)
{
  return prev_meas_gap_start(tti, period, offset) + period;
}
```

Method Name:

tti_point nearest_meas_gap()

Parameters:

tti_point tti

uint32_t period

uint32_t offset

Functionality:

Gets tti point

Returns:

tti_point{static_cast<uint32_t>(round(static_cast<float>((tti - offset).to_uint()) / period)) * period + offset}

Code:

```
tti_point nearest_meas_gap(tti_point tti, uint32_t period, uint32_t offset)
{
  return tti_point{static_cast<uint32_t>(round(static_cast<float>((tti - offset).to_uint()) / period)) * period +
                   offset};
}
```

Method Name:

bool sched_ue::pdsch_enabled() const

Parameters:

srsran::tti_point tti_rx

uint32_t enb_cc_idx

Functionality:

Check CC index is correct

Checks measGap collision

disables TTIs that lead to PDCCH/PDSCH or respective ACKs to fall in measGap

Returns:

True

False

Code:

```
bool sched_ue::pdsch_enabled(srsran::tti_point tti_rx, uint32_t enb_cc_idx) const
{
  if (cfg.supported_cc_list[0].enb_cc_idx != enb_cc_idx) {
    return true;
  }

  // Check measGap collision
  if (cfg.measgap_period > 0) {
    tti_point    tti_tx_dl = to_tx_dl(tti_rx), tti_tx_dl_ack = to_tx_dl_ack(tti_rx);
    tti_point    mgap_tti = nearest_meas_gap(tti_tx_dl, cfg.measgap_period, cfg.measgap_offset);
    tti_interval meas_gap{mgap_tti, mgap_tti + 6};

    // disable TTIs that lead to PDCCH/PDSCH or respective ACKs to fall in measGap
    if (meas_gap.contains(tti_tx_dl) or meas_gap.contains(tti_tx_dl_ack)) {
      return false;
    }
  }
  return true;
}
```

Method Name:

bool sched_ue::pusch_enabled() const

Parameters:

tti_point tti_rx

uint32_t enb_cc_idx

bool needs_pdcch

Functionality:

Checks CC index

Check measGap collision

disable TTIs that leads to PUSCH tx or PHICH rx falling in measGap

disable TTIs which respective PDCCH falls in measGap (in case PDCCH is needed)

Returns:

True

False

Code:

```
bool sched_ue::pusch_enabled(tti_point tti_rx, uint32_t enb_cc_idx, bool needs_pdcch) const
{
  if (cfg.supported_cc_list[0].enb_cc_idx != enb_cc_idx) {
    return true;
  }

  // Check measGap collision
  if (cfg.measgap_period > 0) {
    tti_point    tti_tx_ul = to_tx_ul(tti_rx);
    tti_point    mgap_tti  = nearest_meas_gap(tti_tx_ul, cfg.measgap_period, cfg.measgap_offset);
    tti_interval meas_gap{mgap_tti, mgap_tti + 6};

    // disable TTIs that leads to PUSCH tx or PHICH rx falling in measGap
    if (meas_gap.contains(tti_tx_ul) or meas_gap.contains(to_tx_ul_ack(tti_rx))) {
      return false;
    }
    // disable TTIs which respective PDCCH falls in measGap (in case PDCCH is needed)
    if (needs_pdcch and meas_gap.contains(to_tx_dl(tti_rx))) {
      return false;
    }
  }
  return true;
}
```

Method Name:

int sched_ue::set_ack_info()

Parameters:

tti_point tti_rx

uint32_t enb_cc_idx

uint32_t tb_idx

bool ack

Functionality:

Gets download acknowledgement

Returns:

tbs_acked

Code:

```
int sched_ue::set_ack_info(tti_point tti_rx, uint32_t enb_cc_idx, uint32_t tb_idx, bool ack)
{
  int tbs_acked = -1;
  if (cells[enb_cc_idx].cc_state() != cc_st::idle) {
    std::pair<uint32_t, int> p2 = cells[enb_cc_idx].harq_ent.set_ack_info(tti_rx, tb_idx, ack);
    tbs_acked                   = p2.second;
    if (tbs_acked > 0) {
      logger.debug(
          "SCHED: Set DL ACK=%d for rnti=0x%x, pid=%d, tb=%d, tti=%d", ack, rnti, p2.first, tb_idx, tti_rx.to_uint());
    } else {
      logger.warning("SCHED: Received ACK info for unknown TTI=%d", tti_rx.to_uint());
    }
  } else {
    logger.warning("Received DL ACK for invalid cell index %d", enb_cc_idx);
  }
  return tbs_acked;
}
```

Method Name:

void sched_ue::set_ul_crc()

Parameters:

tti_point tti_rx

uint32_t enb_cc_idx

bool crc_res

Functionality:

Gets upload CRC

Returns:

void

Code:

```
void sched_ue::set_ul_crc(tti_point tti_rx, uint32_t enb_cc_idx, bool crc_res)
{
  if (cells[enb_cc_idx].cc_state() != cc_st::idle) {
    int ret = cells[enb_cc_idx].harq_ent.set_ul_crc(tti_rx, 0, crc_res);
    if (ret < 0) {
      logger.warning("Received UL CRC for invalid tti_rx=%d", (int)tti_rx.to_uint());
    }
  } else {
    logger.warning("Received UL CRC for invalid cell index %d", enb_cc_idx);
  }
}
```

Method Name:

void sched_ue::set_dl_ri()

Parameters:

tti_point tti_rx

uint32_t enb_cc_idx

uint32_t ri

Functionality:

Sets download RI

Returns:

void

Code:

```
void sched_ue::set_dl_ri(tti_point tti_rx, uint32_t enb_cc_idx, uint32_t ri)
{
  if (cells[enb_cc_idx].cc_state() != cc_st::idle) {
    cells[enb_cc_idx].dl_ri        = ri;
    cells[enb_cc_idx].dl_ri_tti_rx = tti_rx;
  } else {
    logger.warning("Received DL RI for invalid cell index %d", enb_cc_idx);
  }
}
```

Method Name:

void sched_ue::set_dl_pmi(t)

Parameters:

ti_point tti_rx

uint32_t enb_cc_idx

uint32_t pmi

Functionality:

Sets download PMI

Returns:

void

Code:

```
void sched_ue::set_dl_pmi(tti_point tti_rx, uint32_t enb_cc_idx, uint32_t pmi)
{
  if (cells[enb_cc_idx].cc_state() != cc_st::idle) {
    cells[enb_cc_idx].dl_pmi        = pmi;
    cells[enb_cc_idx].dl_pmi_tti_rx = tti_rx;
  } else {
    logger.warning("Received DL PMI for invalid cell index %d", enb_cc_idx);
  }
}
```

Method Name:

void sched_ue::set_dl_cqi()

Parameters:

tti_point tti_rx

uint32_t enb_cc_idx

uint32_t cqi

Functionality:

Sets download CQI

Returns:

void

Code:

```
void sched_ue::set_dl_cqi(tti_point tti_rx, uint32_t enb_cc_idx, uint32_t cqi)
{
  if (cells[enb_cc_idx].cc_state() != cc_st::idle) {
    cells[enb_cc_idx].set_dl_cqi(tti_rx, cqi);
  } else {
    logger.warning("Received DL CQI for invalid enb cell index %d", enb_cc_idx);
  }
}
```

Method Name:

void sched_ue::set_ul_snr()

Parameters:

tti_point tti_rx

uint32_t enb_cc_idx

float snr

uint32_t ul_ch_code

Functionality:

Sets upload SNR

Returns:

void

Code:

```
void sched_ue::set_ul_snr(tti_point tti_rx, uint32_t enb_cc_idx, float snr, uint32_t ul_ch_code)
{
  if (cells[enb_cc_idx].cc_state() != cc_st::idle) {
    cells[enb_cc_idx].tpc_fsm.set_snr(snr, ul_ch_code);
    if (ul_ch_code == tpc::PUSCH_CODE) {
      cells[enb_cc_idx].ul_cqi        = srsran_cqi_from_snr(snr);
      cells[enb_cc_idx].ul_cqi_tti_rx = tti_rx;
    }
  } else {
    logger.warning("Received SNR info for invalid cell index %d", enb_cc_idx);
  }
}
```

Method Name:

tbs_info sched_ue::allocate_new_dl_mac_pdu()

Parameters:

sched::dl_sched_data_t* data

dl_harq_proc* h

const rbgmask_t& user_mask

tti_point tti_tx_dl

uint32_t enb_cc_idx

Uint32_t cfi

uint32_t tb

Functionality:

Allocate MAC PDU (subheaders, CEs, and SDUS)

Allocate DL UE Harq

Gets tb information

Returns:

tb_info

Code:

```
tbs_info sched_ue::allocate_new_dl_mac_pdu(sched::dl_sched_data_t* data,
                                           dl_harq_proc*           h,
                                           const rbgmask_t&        user_mask,
                                           tti_point               tti_tx_dl,
                                           uint32_t                enb_cc_idx,
                                           uint32_t                cfi,
                                           uint32_t                tb)
{
  srsran_dci_dl_t* dci     = &data->dci;
  uint32_t         nof_prb = count_prb_per_tb(user_mask);
  tbs_info         tb_info = compute_mcs_and_tbs(enb_cc_idx, tti_tx_dl, nof_prb, cfi, *dci);

  // Allocate MAC PDU (subheaders, CEs, and SDUS)
  int rem_tbs = tb_info.tbs_bytes;
  if (cells[enb_cc_idx].is_pcell()) {
    rem_tbs -= allocate_mac_ces(data, lch_handler, rem_tbs);
  }
  rem_tbs -= allocate_mac_sdus(data, lch_handler, rem_tbs, tb);



  // Allocate DL UE Harq
  if (rem_tbs != tb_info.tbs_bytes) {
    h->new_tx(
        user_mask, tb, tti_tx_dl, tb_info.mcs, tb_info.tbs_bytes, data->dci.location.ncce, get_ue_cfg().maxharq_tx);
  } else {
    // Note: At this point, the allocation of bytes to a TB should not fail, unless the RLC buffers have been
    //       emptied by another allocated tb_idx.
    uint32_t pending_bytes = lch_handler.get_dl_tx_total();
    if (pending_bytes > 0) {
      logger.warning("SCHED: Failed to allocate DL TB with tb_idx=%d, tbs=%d, pid=%d. Pending DL buffer data=%d",
                     tb,
                     rem_tbs,
                     h->get_id(),
                     pending_bytes);
    } else {
      logger.info("SCHED: DL TB tb_idx=%d, tbs=%d, pid=%d did not get allocated.", tb, rem_tbs, h->get_id());
    }
    tb_info.tbs_bytes = 0;
    tb_info.mcs       = 0;
  }

  return tb_info;
}
```

Method Name:

int sched_ue::generate_dl_dci_format()

Parameters:

uint32_t pid

sched_interface::dl_sched_data_t* data

tti_point tti_tx_dl

uint32_t enb_cc_idx

uint32_t cfi

const rbgmask_t& user_mask

Functionality:

Set common DCI fields

Sets DCI format

If allocation successful, encode TPC

Returns:

tbs_bytes

Code:

```
int sched_ue::generate_dl_dci_format(uint32_t                          pid,
                                     sched_interface::dl_sched_data_t* data,
                                     tti_point                         tti_tx_dl,
                                     uint32_t                          enb_cc_idx,
                                     uint32_t                          cfi,
                                     const rbgmask_t&                  user_mask)
{
  srsran_dci_format_t dci_format = get_dci_format();
  int                 tbs_bytes  = 0;

  // Set common DCI fields
  srsran_dci_dl_t* dci = &data->dci;
  dci->rnti            = rnti;
  dci->pid             = pid;
  dci->ue_cc_idx       = cells[enb_cc_idx].get_ue_cc_idx();
  dci->format          = dci_format;


  switch (dci_format) {
    case SRSRAN_DCI_FORMAT1A:
      tbs_bytes = generate_format1a(pid, data, tti_tx_dl, enb_cc_idx, cfi, user_mask);
      break;
    case SRSRAN_DCI_FORMAT1:
      tbs_bytes = generate_format1(pid, data, tti_tx_dl, enb_cc_idx, cfi, user_mask);
      break;
    case SRSRAN_DCI_FORMAT2:
      tbs_bytes = generate_format2(pid, data, tti_tx_dl, enb_cc_idx, cfi, user_mask);
      break;
    case SRSRAN_DCI_FORMAT2A:
      tbs_bytes = generate_format2a(pid, data, tti_tx_dl, enb_cc_idx, cfi, user_mask);
      break;
    default:
      logger.error("DCI format (%d) not implemented", dci_format);
  }

  // If allocation successful, encode TPC
  if (tbs_bytes > 0) {
    dci->tpc_pucch = cells[enb_cc_idx].tpc_fsm.encode_pucch_tpc();
  }

  return tbs_bytes;
}
```

Method Name:

int sched_ue::generate_format1a()

Parameters:

uint32_t pid

sched_interface::dl_sched_data_t* data

tti_point tti_tx_dl

uint32_t enb_cc_idx

uint32_t cfi

const rbgmask_t& user_mask

Functionality:

Generates format 1a for DCI

Returns:

generate_format1_common(pid, data, tti_tx_dl, enb_cc_idx, cfi, user_mask)

Code:

```
int sched_ue::generate_format1a(uint32_t                          pid,
                                sched_interface::dl_sched_data_t* data,
                                tti_point                         tti_tx_dl,
                                uint32_t                          enb_cc_idx,
                                uint32_t                          cfi,
                                const rbgmask_t&                  user_mask)
{
  srsran_dci_dl_t* dci = &data->dci;

  dci->alloc_type       = SRSRAN_RA_ALLOC_TYPE2;
  dci->type2_alloc.mode = srsran_ra_type2_t::SRSRAN_RA_TYPE2_LOC;
  rbg_interval rbg_int  = rbg_interval::rbgmask_to_rbgs(user_mask);
  prb_interval prb_int  = prb_interval::rbgs_to_prbs(rbg_int, cell.nof_prb);
  uint32_t     L_crb    = prb_int.length();
  uint32_t     RB_start = prb_int.start();
  dci->type2_alloc.riv  = srsran_ra_type2_to_riv(L_crb, RB_start, cell.nof_prb);
  if (L_crb != count_prb_per_tb(user_mask)) {
    // This happens if Type0 was using distributed allocation
    logger.warning("SCHED: Can't use distributed RA due to DCI size ambiguity");
  }

  return generate_format1_common(pid, data, tti_tx_dl, enb_cc_idx, cfi, user_mask);
}
```

Method Name:

int sched_ue::generate_format1_common()

Parameters:

uint32_t pid

sched_interface::dl_sched_data_t* data

tti_point tti_tx_dl

uint32_t enb_cc_idx

uint32_t cfi

const rbgmask_t& user_mask

Functionality:

Generates format 1 common for DCI

Returns:

tbinfo.tbs_bytes

Code:

```
int sched_ue::generate_format1_common(uint32_t                          pid,
                                      sched_interface::dl_sched_data_t* data,
                                      tti_point                         tti_tx_dl,
                                      uint32_t                          enb_cc_idx,
                                      uint32_t                          cfi,
                                      const rbgmask_t&                  user_mask)
{
  dl_harq_proc*    h   = &cells[enb_cc_idx].harq_ent.dl_harq_procs()[pid];
  srsran_dci_dl_t* dci = &data->dci;

  tbs_info tbinfo;
  if (h->is_empty(0)) {
    tbinfo = allocate_new_dl_mac_pdu(data, h, user_mask, tti_tx_dl, enb_cc_idx, cfi, 0);
  } else {
    h->new_retx(user_mask, 0, tti_tx_dl, &tbinfo.mcs, &tbinfo.tbs_bytes, dci->location.ncce);
    logger.debug("SCHED: Alloc format1 previous mcs=%d, tbs=%d", tbinfo.mcs, tbinfo.tbs_bytes);
  }

  if (tbinfo.tbs_bytes > 0) {
    dci->tb[0].mcs_idx = (uint32_t)tbinfo.mcs;
    dci->tb[0].rv      = get_rvidx(h->nof_retx(0));
    dci->tb[0].ndi     = h->get_ndi(0);

    data->tbs[0] = (uint32_t)tbinfo.tbs_bytes;
    data->tbs[1] = 0;
  }
  return tbinfo.tbs_bytes;
}
```

Method Name:

int sched_ue::generate_format1()

Parameters:

uint32_t pid

sched_interface::dl_sched_data_t* data

tti_point tti_tx_dl

uint32_t enb_cc_idx

uint32_t cfi

const rbgmask_t& user_mask

Functionality:

Gets mask for format 1

Returns:

generate_format1_common(pid, data, tti_tx_dl, enb_cc_idx, cfi, user_mask)

Code:

```
int sched_ue::generate_format1(uint32_t                          pid,
                               sched_interface::dl_sched_data_t* data,
                               tti_point                         tti_tx_dl,
                               uint32_t                          enb_cc_idx,
                               uint32_t                          cfi,
                               const rbgmask_t&                  user_mask)
{
  srsran_dci_dl_t* dci = &data->dci;

  dci->alloc_type              = SRSRAN_RA_ALLOC_TYPE0;
  dci->type0_alloc.rbg_bitmask = (uint32_t)user_mask.to_uint64();

  return generate_format1_common(pid, data, tti_tx_dl, enb_cc_idx, cfi, user_mask);
}
```

Method Name:

tbs_info sched_ue::compute_mcs_and_tbs()

Parameters:

uint32_t enb_cc_idx

tti_point tti_tx_dl

uint32_t nof_alloc_prbs

uint32_t cfi

const srsran_dci_dl_t& dci

Functionality:

Calculate exact number of RE for this PRB allocation

Compute MCS+TBS

Returns:

tb

Code:

```
tbs_info sched_ue::compute_mcs_and_tbs(uint32_t                enb_cc_idx,
                                       tti_point               tti_tx_dl,
                                       uint32_t                nof_alloc_prbs,
                                       uint32_t                cfi,
                                       const srsran_dci_dl_t& dci)
{
  assert(cells[enb_cc_idx].configured());
  srsran::interval<uint32_t> req_bytes = get_requested_dl_bytes(enb_cc_idx);

  // Calculate exact number of RE for this PRB allocation
  uint32_t nof_re = cells[enb_cc_idx].cell_cfg->get_dl_nof_res(tti_tx_dl, dci, cfi);

  // Compute MCS+TBS
  tbs_info tb = cqi_to_tbs_dl(cells[enb_cc_idx], nof_alloc_prbs, nof_re, dci.format, req_bytes.stop());

  if (tb.tbs_bytes > 0 and tb.tbs_bytes < (int)req_bytes.start()) {
    logger.info("SCHED: Could not get PRB allocation that avoids MAC CE or RLC SRB0 PDU segmentation");
    // Note: This is not a warning, because the srb0 buffer can be updated after the ue sched decision
  }

  return tb;
}
```

Method Name:

int sched_ue::generate_format2a()

Parameters:

uint32_t pid

sched_interface::dl_sched_data_t* data

tti_point tti_tx_dl

uint32_t enb_cc_idx

uint32_t cfi

const rbgmask_t& user_mask

Functionality:

Generates a Format2a DCI

Fill DCI TB dedicated fields

Sets up tb buffers

Returns:

ret

Code:

```
int sched_ue::generate_format2a(uint32_t                          pid,
                                sched_interface::dl_sched_data_t* data,
                                tti_point                         tti_tx_dl,
                                uint32_t                          enb_cc_idx,
                                uint32_t                          cfi,
                                const rbgmask_t&                  user_mask)
{
  dl_harq_proc* h                    = &cells[enb_cc_idx].harq_ent.dl_harq_procs()[pid];
  bool          tb_en[SRSRAN_MAX_TB] = {false};

  srsran_dci_dl_t* dci         = &data->dci;
  dci->alloc_type              = SRSRAN_RA_ALLOC_TYPE0;
  dci->type0_alloc.rbg_bitmask = (uint32_t)user_mask.to_uint64();

  bool no_retx = true;


 if (cells[enb_cc_idx].dl_ri == 0) {
   if (h->is_empty(1)) {
     /* One layer, tb1 buffer is empty, send tb0 only */
     tb_en[0] = true;
   } else {
     /* One layer, tb1 buffer is not empty, send tb1 only */
     tb_en[1] = true;
   }
 } else {
   /* Two layers, retransmit what TBs that have not been Acknowledged */
   for (uint32_t tb = 0; tb < SRSRAN_MAX_TB; tb++) {
     if (!h->is_empty(tb)) {
       tb_en[tb] = true;
       no_retx   = false;
     }
   }
   /* Two layers, no retransmissions...  */
   if (no_retx) {
     tb_en[0] = true;
     tb_en[1] = true;
   }
 }
```

```
  for (uint32_t tb = 0; tb < SRSRAN_MAX_TB; tb++) {
    tbs_info tbinfo;

    if (!h->is_empty(tb)) {
      h->new_retx(user_mask, tb, tti_tx_dl, &tbinfo.mcs, &tbinfo.tbs_bytes, data->dci.location.ncce);
    } else if (tb_en[tb] && no_retx) {
      tbinfo = allocate_new_dl_mac_pdu(data, h, user_mask, tti_tx_dl, enb_cc_idx, cfi, tb);
    }

    /* Fill DCI TB dedicated fields */
    if (tbinfo.tbs_bytes > 0 && tb_en[tb]) {
      dci->tb[tb].mcs_idx = (uint32_t)tbinfo.mcs;
      dci->tb[tb].rv      = get_rvidx(h->nof_retx(tb));
      if (!SRSRAN_DCI_IS_TB_EN(dci->tb[tb])) {
        dci->tb[tb].rv = 2;
      }
      dci->tb[tb].ndi    = h->get_ndi(tb);
      dci->tb[tb].cw_idx = tb;
      data->tbs[tb]      = (uint32_t)tbinfo.tbs_bytes;
    } else {
      SRSRAN_DCI_TB_DISABLE(dci->tb[tb]);
      data->tbs[tb] = 0;
    }
  }

  int ret = data->tbs[0] + data->tbs[1];
  return ret;
}
```

Method Name:

int sched_ue::generate_format2()

Parameters:

uint32_t pid

sched_interface::dl_sched_data_t* data

tti_point tti_tx_dl

uint32_t enb_cc_idx

uint32_t cfi

const rbgmask_t& user_mask

Functionality:

Generates a Format2 DCI

Call Format 2a (common)

Compute precoding information

Returns:

ret

Code:

```
int sched_ue::generate_format2(uint32_t                          pid,
                               sched_interface::dl_sched_data_t* data,
                               tti_point                         tti_tx_dl,
                               uint32_t                          enb_cc_idx,
                               uint32_t                          cfi,
                               const rbgmask_t&                  user_mask)
{
  /* Call Format 2a (common) */
  int ret = generate_format2a(pid, data, tti_tx_dl, enb_cc_idx, cfi, user_mask);

  /* Compute precoding information */
  if ((SRSRAN_DCI_IS_TB_EN(data->dci.tb[0]) + SRSRAN_DCI_IS_TB_EN(data->dci.tb[1])) == 1) {
    data->dci.pinfo = (uint8_t)(cells[enb_cc_idx].dl_pmi + 1) % (uint8_t)5;
  } else {
    data->dci.pinfo = (uint8_t)(cells[enb_cc_idx].dl_pmi & 1u);
  }

  return ret;
}
```

Method Name:

int sched_ue::generate_format0()

Parameters:

sched_interface::ul_sched_data_t* data

tti_point tti_tx_ul

uint32_t enb_cc_idx

prb_interval alloc

bool needs_pdcch

srsran_dci_location_t dci_pos

int explicit_mcs

uci_pusch_t uci_type

Functionality:

Get CQI request

Set DCI position

Get new transmit

Reduce MCS to fit UCI if transmitted in this grant

Add the RE for CQI report

Recompute again the MCS and TBS with the new spectral efficiency

If Msg3 set different number of retx

Un-trigger the SR if data is allocated

If there are no RE available for ULSCH but there is UCI to transmit, allocate PUSCH because resources have been reserved already and in CA it will be used to ACK other carriers

Returns:

tbinfo.tbs_bytes

Code:

```
int sched_ue::generate_format0(sched_interface::ul_sched_data_t* data,
                               tti_point                         tti_tx_ul,
                               uint32_t                          enb_cc_idx,
                               prb_interval                      alloc,
                               bool                              needs_pdcch,
                               srsran_dci_location_t             dci_pos,
                               int                               explicit_mcs,
                               uci_pusch_t                       uci_type)
{
  ul_harq_proc*    h   = get_ul_harq(tti_tx_ul, enb_cc_idx);
  srsran_dci_ul_t* dci = &data->dci;

  bool cqi_request = needs_cqi(tti_tx_ul.to_uint(), enb_cc_idx, true);

  // Set DCI position
  data->needs_pdcch = needs_pdcch;
  dci->location     = dci_pos;

  tbs_info tbinfo;
  tbinfo.mcs       = (explicit_mcs >= 0) ? explicit_mcs : cells[enb_cc_idx].fixed_mcs_ul;
  tbinfo.tbs_bytes = 0;
```

```
bool is_newtx = h->is_empty(0);
if (is_newtx) {
  if (tbinfo.mcs >= 0) {
    tbinfo.tbs_bytes = get_tbs_bytes(tbinfo.mcs, alloc.length(), false, true);
  } else {
    // dynamic mcs
    uint32_t req_bytes = get_pending_ul_new_data(tti_tx_ul, enb_cc_idx);
    uint32_t N_srs     = 0;
    uint32_t nof_symb  = 2 * (SRSRAN_CP_NSYMB(cell.cp) - 1) - N_srs;
    uint32_t nof_re    = nof_symb * alloc.length() * SRSRAN_NRE;
    tbinfo             = cqi_to_tbs_ul(cells[enb_cc_idx], alloc.length(), nof_re, req_bytes);

    // Reduce MCS to fit UCI if transmitted in this grant
    if (uci_type != UCI_PUSCH_NONE) {
      // Calculate an approximation of the number of RE used by UCI
      uint32_t nof_uci_re = 0;
      // Add the RE for ACK
      if (uci_type == UCI_PUSCH_ACK || uci_type == UCI_PUSCH_ACK_CQI) {
        float beta = srsran_sch_beta_ack(cfg.uci_offset.I_offset_ack);
        nof_uci_re +=
            srsran_qprime_ack_ext(alloc.length(), nof_symb, 8 * tbinfo.tbs_bytes, cfg.supported_cc_list.size(), beta);
      }
      // Add the RE for CQI report (RI reports are transmitted on CQI slots. We do a conservative estimate here)
      if (uci_type == UCI_PUSCH_CQI || uci_type == UCI_PUSCH_ACK_CQI || cqi_request) {
        float beta = srsran_sch_beta_cqi(cfg.uci_offset.I_offset_cqi);
        nof_uci_re += srsran_qprime_cqi_ext(alloc.length(), nof_symb, 8 * tbinfo.tbs_bytes, beta);
      }
      // Recompute again the MCS and TBS with the new spectral efficiency (based on the available RE for data)
      if (nof_re >= nof_uci_re) {
        tbinfo = cqi_to_tbs_ul(cells[enb_cc_idx], alloc.length(), nof_re - nof_uci_re, req_bytes);
      }
      // NOTE: if (nof_re < nof_uci_re) we should set TBS=0
    }
  }


  // If Msg3 set different nof retx
  uint32_t nof_retx = (data->needs_pdcch) ? get_max_retx() : max_msg3retx;
  h->new_tx(tti_tx_ul, tbinfo.mcs, tbinfo.tbs_bytes, alloc, nof_retx, not data->needs_pdcch);
  // Un-trigger the SR if data is allocated
  if (tbinfo.tbs_bytes > 0) {
    unset_sr();
  }
} else {
  // retx
  h->new_retx(tti_tx_ul, &tbinfo.mcs, nullptr, alloc);
  tbinfo.tbs_bytes = get_tbs_bytes(tbinfo.mcs, alloc.length(), false, true);
}

if (tbinfo.tbs_bytes >= 0) {
  data->tbs           = tbinfo.tbs_bytes;
  data->current_tx_nb = h->nof_retx(0);
  dci->rnti           = rnti;
  dci->format         = SRSRAN_DCI_FORMAT0;
  dci->ue_cc_idx      = cells[enb_cc_idx].get_ue_cc_idx();
  dci->tb.ndi         = h->get_ndi(0);
  dci->cqi_request    = cqi_request;
  dci->freq_hop_fl    = srsran_dci_ul_t::SRSRAN_RA_PUSCH_HOP_DISABLED;
  dci->tpc_pusch      = cells[enb_cc_idx].tpc_fsm.encode_pusch_tpc();

  dci->type2_alloc.riv = srsran_ra_type2_to_riv(alloc.length(), alloc.start(), cell.nof_prb);
```

```
    // If there are no RE available for ULSCH but there is UCI to transmit, allocate PUSCH becuase
    // resources have been reserved already and in CA it will be used to ACK other carriers
    if (tbinfo.tbs_bytes == 0 && (cqi_request || uci_type != UCI_PUSCH_NONE)) {
      // 8.6.1 and 8.6.2 36.213 second paragraph
      dci->cqi_request = true;
      dci->tb.mcs_idx  = 29;
      dci->tb.rv       = 0; // No data is being transmitted

      // Empty TBS PUSCH only accepts a maximum of 4 PRB. Resize the grant. This doesn't affect the MCS selection
      // because there is no TB in this grant
      if (alloc.length() > 4) {
        alloc.set(alloc.start(), alloc.start() + 4);
      }
    } else if (tbinfo.tbs_bytes > 0) {
      dci->tb.rv = get_rvidx(h->nof_retx(0));
      if (!is_newtx && data->needs_pdcch) {
        dci->tb.mcs_idx = 28 + dci->tb.rv;
      } else {
        dci->tb.mcs_idx = tbinfo.mcs;
      }
    } else if (tbinfo.tbs_bytes == 0) {
      logger.warning("SCHED: No space for ULSCH while allocating format0. Discarding grant.");
    } else {
      logger.error("SCHED: Unkown error while allocating format0");
    }
  }

  return tbinfo.tbs_bytes;
}
```

Method Name:

uint32_t sched_ue::get_max_retx()

Parameters:

none

Functionality:

Gets max retx

Returns:

cfg.maxharq_tx

Code:

```
uint32_t sched_ue::get_max_retx()
{
  return cfg.maxharq_tx;
}
```

Method Name:

bool sched_ue::needs_cqi()

Parameters:

uint32_t tti

uint32_t enb_cc_idx

bool will_send

Functionality:

Sets ret to false

Sets ret to true if configuration is met

Returns:

ret(true/false)

Code:

```
bool sched_ue::needs_cqi(uint32_t tti, uint32_t enb_cc_idx, bool will_send)
{
  bool ret = false;
  if (phy_config_dedicated_enabled && cfg.supported_cc_list[0].aperiodic_cqi_period &&
      lch_handler.has_pending_dl_txs()) {
    uint32_t interval = srsran_tti_interval(tti, cells[enb_cc_idx].dl_cqi_tti_rx.to_uint());
    bool     needscqi = interval >= cfg.supported_cc_list[0].aperiodic_cqi_period;
    if (needscqi) {
      uint32_t interval_sent = srsran_tti_interval(tti, cqi_request_tti);
      if (interval_sent >= 16) {
        if (will_send) {
          cqi_request_tti = tti;
        }
        logger.debug("SCHED: Needs_cqi, last_sent=%d, will_be_sent=%d", cqi_request_tti, will_send);
        ret = true;
      }
    }
  }
  return ret;
}
```

Method Name:

rbg_interval sched_ue::get_required_dl_rbgs()

Parameters:

uint32_t enb_cc_idx

Functionality:

Allocates given PRBs

Gets min and max pending PRBs

Returns:

{min_pending_rbg, max_pending_rbg}

Code:

```
rbg_interval sched_ue::get_required_dl_rbgs(uint32_t enb_cc_idx)
{
  assert(cells[enb_cc_idx].configured());
  const auto*                cellparams = cells[enb_cc_idx].cell_cfg;
  srsran::interval<uint32_t> req_bytes  = get_requested_dl_bytes(enb_cc_idx);
  if (req_bytes == srsran::interval<uint32_t>{0, 0}) {
    return {0, 0};
  }
  int pending_prbs = get_required_prb_dl(cells[enb_cc_idx], to_tx_dl(current_tti), get_dci_format(), req_bytes.start());
  if (pending_prbs < 0) {
    // Cannot fit allocation in given PRBs
    logger.error("SCHED: DL CQI=%d does now allow fitting %d non-segmentable DL tx bytes into the cell bandwidth. "
                 "Consider increasing initial CQI value.",
                 cells[enb_cc_idx].dl_cqi,
                 req_bytes.start());
    return {cellparams->nof_prb(), cellparams->nof_prb()};
  }
  uint32_t min_pending_rbg = cellparams->nof_prbs_to_rbgs(pending_prbs);
  pending_prbs = get_required_prb_dl(cells[enb_cc_idx], to_tx_dl(current_tti), get_dci_format(), req_bytes.stop());
  pending_prbs = (pending_prbs < 0) ? cellparams->nof_prb() : pending_prbs;
  uint32_t max_pending_rbg = cellparams->nof_prbs_to_rbgs(pending_prbs);
  return {min_pending_rbg, max_pending_rbg};
}
```

Method Name:

uint32_t sched_ue::get_pending_dl_bytes()

Parameters:

uint32_t enb_cc_idx

Functionality:

Gets requested download bytes

Returns:

get_requested_dl_bytes(enb_cc_idx).stop()

Code:

```
uint32_t sched_ue::get_pending_dl_bytes(uint32_t enb_cc_idx)
{
  return get_requested_dl_bytes(enb_cc_idx).stop();
}
```

Method Name:

srsran::interval<uint32_t> sched_ue::get_requested_dl_bytes()

Parameters:

uint32_t enb_cc_idx

Functionality:

Set Maximum boundary for CC cells

Add pending CEs

Wait for SRB0 data to be available for Msg4 before scheduling the ConRes CE

Add pending data in remaining RLC buffers

Set Minimum boundary

Returns:

{min_data, max_data}

Code:

```
srsran::interval<uint32_t> sched_ue::get_requested_dl_bytes(uint32_t enb_cc_idx)
{
  assert(cells.at(enb_cc_idx).configured());

  /* Set Maximum boundary */
  if (cells[enb_cc_idx].cc_state() != cc_st::active) {
    return {};
  }

  uint32_t max_data = 0, min_data = 0;
  uint32_t srb0_data = 0, rb_data = 0, sum_ce_data = 0;

  srb0_data = lch_handler.get_dl_tx_total_with_overhead(0);
  // Add pending CEs
  if (cells[enb_cc_idx].is_pcell()) {
    if (srb0_data == 0 and not lch_handler.pending_ces.empty() and
        lch_handler.pending_ces.front() == srsran::dl_sch_lcid::CON_RES_ID) {
      // Wait for SRB0 data to be available for Msg4 before scheduling the ConRes CE
      return {};
    }
    for (const lch_ue_manager::ce_cmd& ce : lch_handler.pending_ces) {
      sum_ce_data += srsran::ce_total_size(ce);
    }
  }
  // Add pending data in remaining RLC buffers
  for (int i = 1; i < sched_interface::MAX_LC; i++) {
    rb_data += lch_handler.get_dl_tx_total_with_overhead(i);
  }
  max_data = srb0_data + sum_ce_data + rb_data;


  /* Set Minimum boundary */
  min_data = srb0_data;
  if (not lch_handler.pending_ces.empty() and lch_handler.pending_ces.front() == lch_ue_manager::ce_cmd::CON_RES_ID) {
    min_data += srsran::ce_total_size(lch_handler.pending_ces.front());
  }
  if (min_data == 0) {
    if (sum_ce_data > 0) {
      min_data = srsran::ce_total_size(lch_handler.pending_ces.front());
    } else if (rb_data > 0) {
      min_data = MAC_MIN_ALLOC_SIZE;
    }
  }

  return {min_data, max_data};
}
```

Method Name:

uint32_t sched_ue::get_pending_dl_rlc_data() const

Parameters:

none

Functionality:

Get pending RLC DL data in RLC buffers

Returns:

lch_handler.get_dl_tx_total()

Code:

```
uint32_t sched_ue::get_pending_dl_rlc_data() const
{
  return lch_handler.get_dl_tx_total();
}
```

Method Name:

uint32_t sched_ue::get_expected_dl_bitrate() const

Parameters:

uint32_t enb_cc_idx

int nof_rbgs

Functionality:

Gets tti duration in ms for download

Returns:

tbs / tti_duration_ms

Code:

```
uint32_t sched_ue::get_expected_dl_bitrate(uint32_t enb_cc_idx, int nof_rbgs) const
{
  auto&    cc = cells[enb_cc_idx];
  uint32_t nof_re =
      cc.cell_cfg->get_dl_lb_nof_re(to_tx_dl(current_tti), count_prb_per_tb_approx(nof_rbgs, cc.cell_cfg->nof_prb()));
  float max_coderate = srsran_cqi_to_coderate(std::min(cc.dl_cqi + 1u, 15u), cfg.use_tbs_index_alt);

  // Inverse of srsran_coderate(tbs, nof_re)
  uint32_t tbs = max_coderate * nof_re - 24;
  return tbs / tti_duration_ms;
}
```

Method Name:

uint32_t sched_ue::get_expected_ul_bitrate() const

Parameters:

uint32_t enb_cc_idx

int nof_prbs

Functionality:

Get tti duration in ms for upload

Returns:

tbs / tti_duration_ms

Code:

```
uint32_t sched_ue::get_expected_ul_bitrate(uint32_t enb_cc_idx, int nof_prbs) const
{
  uint32_t nof_prbs_alloc = nof_prbs < 0 ? cell.nof_prb : nof_prbs;

  uint32_t N_srs        = 0;
  uint32_t nof_symb     = 2 * (SRSRAN_CP_NSYMB(cell.cp) - 1) - N_srs;
  uint32_t nof_re       = nof_symb * nof_prbs_alloc * SRSRAN_NRE;
  float    max_coderate = srsran_cqi_to_coderate(std::min(cells[enb_cc_idx].ul_cqi + 1u, 15u), false);

  // Inverse of srsran_coderate(tbs, nof_re)
  uint32_t tbs = max_coderate * nof_re - 24;
  return tbs / tti_duration_ms;
}
```

Method Name:

uint32_t sched_ue::get_pending_ul_old_data()

Parameters:

uint32_t enb_cc_idx

Functionality:

Get pending upload data

Returns:

pending_data

Code:

```
uint32_t sched_ue::get_pending_ul_old_data(uint32_t enb_cc_idx)
{
  uint32_t pending_data = 0;
  for (auto& h : cells[enb_cc_idx].harq_ent.ul_harq_procs()) {
    pending_data += h.get_pending_data();
  }
  return pending_data;
}
```

Method Name:

uint32_t sched_ue::get_pending_ul_old_data()

Parameters:

none

Functionality:

Returns the total of all TB bytes allocated to UL HARQs

Returns:

pending_ul_data

Code:

```
uint32_t sched_ue::get_pending_ul_old_data()
{
  uint32_t pending_ul_data = 0;
  for (uint32_t i = 0; i < cells.size(); ++i) {
    if (cells[i].configured()) {
      pending_ul_data += get_pending_ul_old_data(i);
    }
  }
  return pending_ul_data;
}
```

Method Name:

uint32_t sched_ue::get_pending_ul_data_total()

Parameters:

tti_point tti_tx_ul

int this_enb_cc_idx

Functionality:

Gets pending data from upload

Returns:

512

128

pending_data

Code:

```
uint32_t sched_ue::get_pending_ul_data_total(tti_point tti_tx_ul, int this_enb_cc_idx)
{
  static constexpr uint32_t lbsr_size = 4, sbsr_size = 2;

  // Note: If there are no active bearers, scheduling requests are also ignored.
  uint32_t pending_data = 0;
  uint32_t pending_lcgs = 0;
  for (int lcg = 0; lcg < sched_interface::MAX_LC_GROUP; lcg++) {
    uint32_t bsr = lch_handler.get_bsr_with_overhead(lcg);
    if (bsr > 0) {
      pending_data += bsr;
      pending_lcgs++;
    }
  }
```

```
  if (pending_data > 0) {
    // The scheduler needs to account for the possibility of BSRs being allocated in the UL grant.
    // Otherwise, the UL grants allocated for very small RRC messages (e.g. rrcReconfigurationComplete)
    // may be fully occupied by a BSR, and RRC the message transmission needs to be postponed.
    pending_data += (pending_lcgs <= 1) ? sbsr_size : lbsr_size;
  } else {
    if (is_sr_triggered() and this_enb_cc_idx >= 0) {
      // Check if this_cc_idx is the carrier with highest CQI
      uint32_t max_cqi = 0, max_cc_idx = 0;
      for (uint32_t cc = 0; cc < cells.size(); ++cc) {
        if (cells[cc].configured()) {
          uint32_t sum_cqi = cells[cc].dl_cqi + cells[cc].ul_cqi;
          if (cells[cc].cc_state() == cc_st::active and sum_cqi > max_cqi) {
            max_cqi    = sum_cqi;
            max_cc_idx = cc;
          }
        }
      }
      if ((int)max_cc_idx == this_enb_cc_idx) {
        return 512;
      }
    }
    for (uint32_t i = 0; i < cells.size(); ++i) {
      if (cells[i].configured() and needs_cqi(tti_tx_ul.to_uint(), i)) {
        return 128;
      }
    }
  }

  return pending_data;
}
```

Method Name:

uint32_t sched_ue::get_pending_ul_new_data()

Parameters:

tti_point tti_tx_ul

int this_enb_cc_idx

Functionality:

Subtract all the UL data already allocated in the UL harqs

Gets pending data

Returns:

pending_data

Code:

```
uint32_t sched_ue::get_pending_ul_new_data(tti_point tti_tx_ul, int this_enb_cc_idx)
{
  uint32_t pending_data = get_pending_ul_data_total(tti_tx_ul, this_enb_cc_idx);

  // Subtract all the UL data already allocated in the UL harqs
  uint32_t pending_ul_data = get_pending_ul_old_data();
  pending_data             = (pending_data > pending_ul_data) ? pending_data - pending_ul_data : 0;

  if (pending_data > 0) {
    if (logger.debug.enabled()) {
      fmt::memory_buffer str_buffer;
      fmt::format_to(str_buffer, "{}", lch_handler.get_bsr_state());
      logger.debug("SCHED: pending_data=%d, in_harq_data=%d, bsr=%s",
                   pending_data,
                   pending_ul_data,
                   srsran::to_c_str(str_buffer));
    }
  }
  return pending_data;
}
```

Method Name:

uint32_t sched_ue::get_required_prb_ul()

Parameters:

uint32_t enb_cc_idx

uint32_t req_bytes

Functionality:

Gets upload required PRBs

Returns:

srsenb::get_required_prb_ul(cells[enb_cc_idx], req_bytes)

Code:

```
uint32_t sched_ue::get_required_prb_ul(uint32_t enb_cc_idx, uint32_t req_bytes)
{
  return srsenb::get_required_prb_ul(cells[enb_cc_idx], req_bytes);
}
```

Method Name:

bool sched_ue::is_sr_triggered()

Parameters:

none

Functionality:

Checks if SR is triggered

Returns:

sr(true/false)

Code:

```
bool sched_ue::is_sr_triggered()
{
  return sr;
}
```

Method Name:

dl_harq_proc* sched_ue::get_pending_dl_harq()

Parameters:

tti_point tti_tx_dl

uint32_t enb_cc_idx

Functionality:

Gets HARQ process with oldest pending retx

Returns:

cells[enb_cc_idx].harq_ent.get_pending_dl_harq(tti_tx_dl)

nullptr

Code:

```
dl_harq_proc* sched_ue::get_pending_dl_harq(tti_point tti_tx_dl, uint32_t enb_cc_idx)
{
  if (cells[enb_cc_idx].cc_state() == cc_st::active) {
    return cells[enb_cc_idx].harq_ent.get_pending_dl_harq(tti_tx_dl);
  }
  return nullptr;
}
```

Method Name:

dl_harq_proc* sched_ue::get_empty_dl_harq()

Parameters:

tti_point tti_tx_dl

uint32_t enb_cc_idx

Functionality:

Gets empty HARQ process

Returns:

cells[enb_cc_idx].harq_ent.get_empty_dl_harq(tti_tx_dl)

nullptr

Code:

```
dl_harq_proc* sched_ue::get_empty_dl_harq(tti_point tti_tx_dl, uint32_t enb_cc_idx)
{
  if (cells[enb_cc_idx].cc_state() == cc_st::active) {
    return cells[enb_cc_idx].harq_ent.get_empty_dl_harq(tti_tx_dl);
  }
  return nullptr;
}
```

Method Name:

ul_harq_proc* sched_ue::get_ul_harq()

Parameters:

tti_point tti_tx_ul

uint32_t enb_cc_idx

Functionality:

Gets upload HARQ process

Returns:

cells[enb_cc_idx].harq_ent.get_ul_harq(tti_tx_ul)

nullptr

Code:

```
ul_harq_proc* sched_ue::get_ul_harq(tti_point tti_tx_ul, uint32_t enb_cc_idx)
{
  if (cells[enb_cc_idx].cc_state() == cc_st::active) {
    return cells[enb_cc_idx].harq_ent.get_ul_harq(tti_tx_ul);
  }
  return nullptr;
}
```

Method Name:

const dl_harq_proc& sched_ue::get_dl_harq() const

Parameters:

uint32_t idx

uint32_t enb_cc_idx

Functionality:

Gets download harq

Returns:

cells[enb_cc_idx].harq_ent.dl_harq_procs()[idx]

Code:

```
const dl_harq_proc& sched_ue::get_dl_harq(uint32_t idx, uint32_t enb_cc_idx) const
{
  return cells[enb_cc_idx].harq_ent.dl_harq_procs()[idx];
}
```

Method Name:

std::pair<bool, uint32_t> sched_ue::get_active_cell_index() const

Parameters:

uint32_t enb_cc_idx

Functionality:

Gets index of active cell

Returns:

{false, std::numeric_limits<uint32_t>::max()}

{cells[enb_cc_idx].cc_state() == cc_st::active, cells[enb_cc_idx].get_ue_cc_idx()}

Code:

```
std::pair<bool, uint32_t> sched_ue::get_active_cell_index(uint32_t enb_cc_idx) const
{
  if (cells[enb_cc_idx].configured()) {
    return {cells[enb_cc_idx].cc_state() == cc_st::active, cells[enb_cc_idx].get_ue_cc_idx()};
  }
  return {false, std::numeric_limits<uint32_t>::max()};
}
```

Method Name:

uint32_t sched_ue::get_aggr_level()

Parameters:

uint32_t enb_cc_idx

uint32_t nof_bits

Functionality:

Get aggr level

Returns:

srsenb::get_aggr_level(nof_bits,cc.dl_cqi,cc.max_aggr_level,cc.cell_cfg->nof_prb(), cfg.use_tbs_index_alt)

Code:

```
uint32_t sched_ue::get_aggr_level(uint32_t enb_cc_idx, uint32_t nof_bits)
{
  const auto& cc = cells[enb_cc_idx];
  return srsenb::get_aggr_level(nof_bits, cc.dl_cqi, cc.max_aggr_level, cc.cell_cfg->nof_prb(), cfg.use_tbs_index_alt);
}
```

Method Name:

void sched_ue::finish_tti()

Parameters:

tti_point tti_rx

uint32_t enb_cc_idx

Functionality:

Check that scell state needs to change

Returns:

void

Code:

```
void sched_ue::finish_tti(tti_point tti_rx, uint32_t enb_cc_idx)
{
  // Check that scell state needs to change
  cells[enb_cc_idx].finish_tti(tti_rx);
}
```

Method Name:

srsran_dci_format_t sched_ue::get_dci_format()

Parameters:

none

Functionality:

Gets the DCI format

Returns:

ret

Code:

```
srsran_dci_format_t sched_ue::get_dci_format()
{
  srsran_dci_format_t ret = SRSRAN_DCI_FORMAT1A;

  if (phy_config_dedicated_enabled) {
    /* TODO: Assumes UE-Specific Search Space (Not common) */
    switch (cfg.dl_ant_info.tx_mode) {
      case sched_interface::ant_info_ded_t::tx_mode_t::tm1:
      case sched_interface::ant_info_ded_t::tx_mode_t::tm2:
        // If the size of Format1 and Format1A is ambiguous in the common SS, use Format1A since the UE assumes
        // Common SS when spaces collide
        if (not(cell.nof_prb == 15 && cells.size() > 1)) {
          ret = SRSRAN_DCI_FORMAT1;
        }
        break;
      case sched_interface::ant_info_ded_t::tx_mode_t::tm3:
        ret = SRSRAN_DCI_FORMAT2A;
        break;
      case sched_interface::ant_info_ded_t::tx_mode_t::tm4:
        ret = SRSRAN_DCI_FORMAT2;
        break;
      case sched_interface::ant_info_ded_t::tx_mode_t::tm5:
      case sched_interface::ant_info_ded_t::tx_mode_t::tm6:
      case sched_interface::ant_info_ded_t::tx_mode_t::tm7:
      case sched_interface::ant_info_ded_t::tx_mode_t::tm8_v920:
      default:
        logger.warning(
            "Incorrect transmission mode (rnti=%04x; tm=%d)", rnti, static_cast<int>(cfg.dl_ant_info.tx_mode));
    }
  }

  return ret;
}
```

Method Name:

const cce_cfi_position_table* sched_ue::get_locations() const

Parameters:

uint32_t enb_cc_idx

uint32_t cfi

uint32_t sf_idx

Functionality:

Gets DCI locations

Returns:

&cells[enb_cc_idx].dci_locations[sf_idx][cfi - 1]

&cells[enb_cc_idx].dci_locations[sf_idx][0]

Code:

```
const cce_cfi_position_table* sched_ue::get_locations(uint32_t enb_cc_idx, uint32_t cfi, uint32_t sf_idx) const
{
  if (cfi > 0 && cfi <= 3) {
    return &cells[enb_cc_idx].dci_locations[sf_idx][cfi - 1];
  } else {
    logger.error("SCHED: Invalid CFI=%d", cfi);
    return &cells[enb_cc_idx].dci_locations[sf_idx][0];
  }
}
```

Method Name:

sched_ue_cell* sched_ue::find_ue_carrier()

Parameters:

uint32_t enb_cc_idx

Functionality:

Gets UE carrier

Returns:

cells[enb_cc_idx].configured() ? &cells[enb_cc_idx] : nullptr

Code:

```
sched_ue_cell* sched_ue::find_ue_carrier(uint32_t enb_cc_idx)
{
  return cells[enb_cc_idx].configured() ? &cells[enb_cc_idx] : nullptr;
}
```

Method Name:

std::bitset<SRSRAN_MAX_CARRIERS> sched_ue::scell_activation_mask() const

Parameters:

none

Functionality:

Sets ret from cc index

Returns:

ret

Code:

```
std::bitset<SRSRAN_MAX_CARRIERS> sched_ue::scell_activation_mask() const
{
  std::bitset<SRSRAN_MAX_CARRIERS> ret{0};
  for (size_t i = 0; i < cells.size(); ++i) {
    if (cells[i].cc_state() == cc_st::active and cells[i].is_scell()) {
      ret[cells[i].get_ue_cc_idx()] = true;
    }
  }
  return ret;
}
```

Method Name:

int sched_ue::enb_to_ue_cc_idx() const

Parameters:

uint32_t enb_cc_idx

Functionality:

Sets enb to CC index

Returns:

enb_cc_idx < cells.size() ? cells[enb_cc_idx].get_ue_cc_idx() : -1

Code:

```
int sched_ue::enb_to_ue_cc_idx(uint32_t enb_cc_idx) const
{
  return enb_cc_idx < cells.size() ? cells[enb_cc_idx].get_ue_cc_idx() : -1;
}
```