ViskaSAT

Tamego

目次

第 1 章
共通インターフェース

1.1 基本設計

SAT ソルバのアルゴリズム部分だけを取り出してライブラリとして使えるようにする。

- **`Solver`** 様々なアルゴリズムのソルバの共通のインターフェース。このトレイトを持つものはロジックだけに集中し、Godot については何も触らない。
- **`SolverRunner`** `Solver` と Godot の橋渡し的役割をする。`Solver` を別スレッドで走らせ、ソルバの進捗を得たりソルバを制御したりと双方向のやりとりを可能にする。

可読性のためにソースを分割して記述する。

```
//| file: rust/viska-sat/src/lib.rs
pub mod lit;
pub mod clause;
pub mod cnf;
pub mod assignment;
pub mod event_handler;
pub mod solver;
pub mod solver_communicator;
```

```
pub mod solver_runner;
pub mod brute_force;
pub mod dpll;
```

1.2　基本的な型

1.2.1　割り当て

各変数に真偽値を対応付ける構造を**割り当て**という。全ての変数に真偽値が割り当てられているとき、その割り当ては**完全**であるという。真・偽・未割り当ての 3 つの値を取り得るので、`Option<bool>` 型を取ることで対応する。 それの配列として割り当てを表現する。 つまり、割り当てが完全なとき、各要素は `Some(true)` か `Some(false)` のいずれかである。

```
//| file: rust/viska-sat/src/assignment.rs
#[derive(Debug, Clone)]
pub struct Assignment {
    pub values: Vec<Option<bool>>
}

impl Assignment {
    pub fn is_full(&self) -> bool {
        for val in &self.values {
            if val.is_none() {
                return false;
            }
        }
        return true;
```

```
    }
}
```

1.2.2 リテラル

リテラルとは原子論理式それ自体もしくはその否定のどちらかである。とくに、原子論理式のときを**正リテラル**と言い、そうでないときは**負リテラル**という。また、命題論理において原子論理式は命題変数だから、リテラルとは命題変数それ自体かその否定のいずれかである。

変数が真に割り当てられたときに正リテラルは充足し、偽に割り当てられたときに負リテラルは充足する。

`var` が命題変数を表し、`negated` が否定されているかどうかを表す。`assign` を取って、以下のいずれかに評価する。

Satisfied 充足

Unsatisfied 非充足

Unassigned 未割り当て

```
//| file: rust/viska-sat/src/lit.rs
use crate::assignment::Assignment;

#[derive(Debug, Clone)]
pub struct Lit {
    pub var_id: usize,
    pub negated: bool
}

pub enum LitState {
    Satisfied,
    Unsatisfied,
    Unassigned
```

```
}

impl Lit {
    pub fn eval(&self, assign: &Assignment) -> LitState {
        match assign.values[self.var_id] {
            None => LitState::Unassigned,
            Some(val) => if val ^ self.negated
{LitState::Satisfied} else {LitState::Unsatisfied}
        }
    }
}
```

1.2.3　節

リテラルを OR で繋いだ論理式を**節**という。`Lit` の配列として表現する。 CDCL ソルバのために必要ならメタ情報を付けることを可能にした。

節内のリテラルのいずれかが充足していれば、その節は充足する。

Satisfied　充足

Unsatisfied　非充足

Unit　単位節

Unresolved　未割り当てが複数個

```
//| file: rust/viska-sat/src/clause.rs
use crate::{assignment::Assignment, lit::{LitState, Lit}};
#[derive(Debug, Clone)]
pub struct Clause<Meta=()> {
    pub lits: Vec<Lit>,
    pub meta: Meta,
}
```

```
pub enum ClauseState {
    Satisfied,
    Unsatisfied,
    Unit(Lit) ,
    Unresolved
}

impl Clause {
    pub fn eval(&self, assign: &Assignment) -> ClauseState {
        let mut all_unsatisfied = true;
        let mut unit_lit = None;
        for lit in &self.lits {
            match lit.eval(assign) {
                LitState::Satisfied => return
ClauseState::Satisfied,
                LitState::Unassigned => {
                    all_unsatisfied = false;
                    if unit_lit.is_some() {
                        return ClauseState::Unresolved;
                    }
                    unit_lit = Some(lit.clone());
                }
                _ => {}
            }
        }

        if all_unsatisfied {
            ClauseState::Unsatisfied
        } else if let Some(lit) = unit_lit {
            ClauseState::Unit(lit)
        } else {
            ClauseState::Unresolved
        }
```

```
    }
}
```

1.2.4　CNF（連言標準形）

節を AND で繋いだ構造をしている論理式を **CNF** という。`Clause` の配列として表現する。 `num_vars` は変数の個数（最大の ID + 1）を表す。 全ての節が充足するときに充足する。

ClauseState が Unit であるような節を集めるメソッドを用意した。

```
//| file: rust/viska-sat/src/cnf.rs
use crate::{assignment::Assignment, clause::{Clause,
ClauseState}};

#[derive(Debug, Clone)]
pub struct Cnf {
    pub clauses: Vec<Clause>,
    pub num_vars: usize
}

#[derive(Debug, Clone)]
pub enum CnfState {
      Satisfied,
      Unsatisfied,
      Unresolved,
}

impl Cnf {
    pub fn eval(&self, assign: &Assignment) -> CnfState {
        let mut all_satisfied = true;
        for clause in &self.clauses {
            match clause.eval(assign) {
```

```
                ClauseState::Unresolved => all_satisfied =
false,
                ClauseState::Unsatisfied => return
CnfState::Unsatisfied,
                _ => {}
            }
        }
        if all_satisfied {
            CnfState::Satisfied
        } else {
            CnfState::Unresolved
        }
    }
}
```

1.3 Solver トレイト

様々なアルゴリズムに共通するインターフェースを与えるために Solver トレイトを定める。 solve() は問題を解き、充足可能（SatResult::Sat）か充足不能（SatResult::Unsat）かをを返す。

SolverReulst は列挙型で定義する。

```
//| id: sol_solver-result
#[derive(Debug, Clone)]
pub enum SatResult {
    Sat(Assignment),
    Unsat
}
```

`Solver` トレイトを定義する。 イベントの種類はソルバ依存なので、`Event` と関連型にして各ソルバが自由に決められるようにした。`EventHandler` については 1.5 節を参照のこと。

```
//| id: sol_solver-trait
pub trait Solver {
    type Event;
    type Error;
    type Handler: EventHandler<Event = Self::Event, Error =
Self::Error>;

    fn solve(&mut self) -> Result<SatResult, Self::Error>;
}
```

```
//| file: rust/viska-sat/src/solver.rs
use crate::{assignment::Assignment,
event_handler::EventHandler};
<<sol_solver-result>>
<<sol_solver-trait>>
```

1.4 `Solver` と `SolverRunner` 間の通信

`Solver` と `SolverRunner` 間の通信のために `SolverCommunicator` を定める。 これは以下の関数を持つ：

`new()` `event_tx` と `ctrl_rx` を引数にとるコンストラクタ。

`send_event()` `SolverRunner` にソルバのイベントを伝える。

`try_recv_latest_control()` `SolverRunner` からの**最新の制御のみ**を受けとる。スレッドをブロックしない。

`recv_latest_control()` `SolverRunner` からの**最新の制御のみ**（`SolverControl`）を受けとる。スレッドをブロックする。

`SolverControl` は全ソルバに共通するので、ここで列挙型で定義する。 停止と再開のみを想定しているが、今後増やすかもしれない。

```
//| id: sol_solver-control
#[derive(PartialEq, Eq)]
pub enum SolverControl {
    Pause,
    Resume
}
```

また、`SolverCommunicator` に関連するエラーハンドリングのための列挙型を作る。

```
//| id: sol_solver-communicator-error
#[derive(Debug)]
pub enum SolverCommunicatorError {
    SendFailed,
    ReceiveFailed,
}
```

イベントの種類はソルバ依存なので、ジェネリクスにして処理する。通信には `mpsc` を使う。 `event_tx` はイベントを `SolverRunner` に知らせ、 `ctrl_rx` で `SolverRunner` からの制御を受け取る。

```
//| id: sol_solver-communicator-decl
pub struct SolverCommunicator<Event> {
    event_tx: Sender<Event>,
    ctrl_rx: Receiver<SolverControl>,
}
```

前述の通りに実装をしていく。

```
//| id: sol_solver-communicator-impl
impl<Event> SolverCommunicator<Event> {
```

```
    <<solsc_constructor>>
    <<solsc_send-event>>
    <<solsc_try-recv-latest-control>>
    <<solsc_recv-latest-control>>
}
```

```
//| id: solsc_constructor
pub fn new(event_tx: Sender<Event>, ctrl_rx:
Receiver<SolverControl>) -> Self {
    Self { event_tx, ctrl_rx }
}
```

イベントを送信する。

```
//| id: solsc_send-event
pub fn send_event(&mut self, event: Event) -> Result<(),
SolverCommunicatorError> {
    if self.event_tx.send(event).is_err() {
        return Err(SolverCommunicatorError::SendFailed);
    }
    Ok(())
}
```

最新の制御メッセージを受け取る。 ここではブロックしない `try_recv()` を使う。 キューが空になるまで最新のメッセージを読むということをやっている。

```
//| id: solsc_try-recv-latest-control
pub fn try_recv_latest_control(&mut self) ->
Result<Option<SolverControl>, SolverCommunicatorError> {
    let mut recv = None;
    loop {
        match self.ctrl_rx.try_recv() {
```

```
            Ok(received) => recv = Some(received),
            Err(TryRecvError::Empty) => break Ok(recv),
            Err(TryRecvError::Disconnected) => return
Err(SolverCommunicatorError::ReceiveFailed),
        }
    }
}
```

今度は `try_recv_latest_control()` のブロックする版 `recv_latest_control()` を定義する。最初の一件だけ `recv()` でブロックして、それに続くメッセージは `try_recv()` で空になるまで見る。後者については `try_recv_latest_control()` そのものなので再利用する。

```
//| id: solsc_recv-latest-control
pub fn recv_latest_control(&mut self) ->
Result<SolverControl, SolverCommunicatorError> {
    let mut recv= match self.ctrl_rx.recv() {
        Ok(val) => val,
        Err(_) => return
Err(SolverCommunicatorError::ReceiveFailed),
    };
    match self.try_recv_latest_control() {
        Ok(Some(received)) => {
            recv = received;
        },
        Err(_) => return
Err(SolverCommunicatorError::ReceiveFailed),
        _ => {}
    }
    Ok(recv)
}
```

```
//| file: rust/viska-sat/src/solver_communicator.rs
use std::sync::mpsc::{Sender, Receiver, TryRecvError};
<<sol_solver-control>>
<<sol_solver-communicator-error>>

<<sol_solver-communicator-decl>>
<<sol_solver-communicator-impl>>
```

1.5　イベントハンドラー

`Solver` のイベントを処理する `EventHandler` トレイトを定義する。`handle_event()` でイベントを処理する。

```
//| id: evh_event-handler-trait
pub trait EventHandler {
    type Event;
    type Error;

    fn handle_event(&mut self, event: Self::Event) ->
Result<(), Self::Error>;
}
```

```
//| file: rust/viska-sat/src/event_handler.rs
<<evh_event-handler-trait>>
```

1.6　`SolverRunner`

別スレッドでソルバを動かし、その進捗をチャネルを通して受けとる。

必要なものを読み込む。

```
//| id: sor_modules
use crate::{event_handler::EventHandler, solver::{SatResult,
Solver}, solver_communicator::{SolverCommunicator,
SolverCommunicatorError, SolverControl}};
use std::sync::mpsc::{channel, Sender, Receiver,
TryRecvError};
use std::thread;
use std::fmt::Debug;
```

1.6.1 イベントハンドラ

まずは、専用のイベントハンドラを定義する。コミュニケータを持ち、ソルバとランナー間のやり取りを実現できるようにする。ソルバによってイベントの型が異なるので、それはジェネリック型で表現する。

```
//| id: sor_solver-runner-event-handler
pub struct SolverRunnerEventHandler<Event> {
    com: SolverCommunicator<Event>,
}

<<soreh_impl>>
```

`handle_event()` を実装する。ここで、関連型について、`Error` は今回使うコミュニケータのエラー型 `CommunicatorError` を使う。

```
//| id: soreh_impl
impl<Event> EventHandler for SolverRunnerEventHandler<Event>
{
    type Event = Event;
    type Error = SolverCommunicatorError;
```

```
    <<soreh_handle-event>>
}
```

以下の様な流れで処理する。

1. もし制御が届いていたら、最新のものに基づいて、`is_pause` を設定する。
2. `is_pause` が真である限りループさせる。これによってソルバ自体をブロックする。
   - 新しく制御が届いていたら、それに基づいて `is_pause` を設定する。
3. イベントを送信する。

```
//| id: soreh_handle-event
fn handle_event(&mut self, event: Self::Event) -> Result<(),
Self::Error> {
    let mut is_pause = false;
    <<soreh_get-latest-control>>
    <<soreh_pause-loop>>
    <<soreh_send-event>>
    Ok(())
}
```

最新の制御が届いていたら、それに基づいて `is_pause` を設定する。現時点では `SolverControl` は `Pause` か `Resume` しかないので、`Pause` であるかどうかで判断している。ここでは非ブロッキング版の `try_recv_latest_control` を使っている。

```
//| id: soreh_get-latest-control
match self.com.try_recv_latest_control() {
    Ok(Some(receive)) => {
        is_pause = receive == SolverControl::Pause;
```

```
    }
    Err(err) => return Err(err),
    _ => {}
};
```

`SolverControl::Resume` が届くまでずっとループする。 ただ、`is_pause` を前述のように設定する方法でも同じことができるので、そのように実装した。 ここではブロッキング版の `recv_latest_control` を使っている。

```
//| id: soreh_pause-loop
while is_pause {
    match self.com.recv_latest_control() {
        Ok(receive) => {
            is_pause = receive == SolverControl::Pause;
        }
        Err(err) => return Err(err),
    }
}
```

イベントを送信する。

```
//| id: soreh_send-event
if let Err(err) = self.com.send_event(event) {
    return Err(err);
}
```

1.6.2 ランナー

スレッドを立てて、そこでソルバを実行する。

`start_solver()` ソルバを別スレッドで走らせ、ランナー自体を返す。

`try_recv_event()`　ソルバからイベントをノンブロッキングで受け取る。

`send_control()`　制御をソルバに伝える。

`try_join()`　スレッドが終了しているかを確認し、もし終了しているならそのスレッドの返り値を返す。

```
//| id: sor_runner
pub struct SolverRunner<S: Solver> {
    event_rx: Receiver<S::Event>,
    ctrl_tx: Sender<SolverControl>,
    join_handle: Option<thread::JoinHandle<Result<SatResult,
S::Error>>>
}

impl<S> SolverRunner<S>
where
    S: Solver + Send,
    S::Event: Send + 'static,
    S::Error: Debug + Send + 'static
{
    <<sorr_start-solver>>
    <<sorr_try-recv-event>>
    <<sorr_send-control>>
    <<sorr_try-join>>
}
```

スレッドを立てて、そこでソルバを実行する。クロージャを使うことで、柔軟にソルバを初期化できるようにしている。ハンドラはこちらで用意するので、ハンドラを受けとってソルバを返すクロージャを取る。

```
//| id: sorr_start-solver
pub fn start_solver<F>(make_solver: F) -> Self
where
    F: (FnOnce(SolverRunnerEventHandler<S::Event>) -> S) +
Send + 'static
{
    let (event_tx, event_rx) = channel::<S::Event>();
    let (ctrl_tx, ctrl_rx) = channel::<SolverControl>();
    let handler = SolverRunnerEventHandler {
        com: SolverCommunicator::new(event_tx, ctrl_rx),
    };
    let join_handle = thread::spawn(move || {
        make_solver(handler).solve()
    });
    Self {
        event_rx,
        ctrl_tx,
        join_handle: Some(join_handle)
    }
}
```

イベントの送受信に関するエラーの列挙型を用意する。

```
//| id: sor_error
#[derive(Debug)]
pub enum SolverRunnerError<E> {
    SendFailed,
    ReceiveFailed,
    SolverError(E),
    NotFinished,
    JoinPanicked,
    AlreadyJoined
}
```

最新のイベントを受けとる。

```
//| id: sorr_try-recv-event
pub fn try_recv_event(&self) -> Result<Option<S::Event>,
SolverRunnerError<S::Error>> {
    match self.event_rx.try_recv() {
        Ok(recv) => return Ok(Some(recv)),
        Err(TryRecvError::Empty) => return Ok(None),
        Err(TryRecvError::Disconnected) => return
Err(SolverRunnerError::ReceiveFailed),
    };
}
```

イベントを送信する。

```
//| id: sorr_send-control
pub fn send_control(&self, control: SolverControl) ->
Result<(), SolverRunnerError<S::Error>> {
    if self.ctrl_tx.send(control).is_err() {
        return Err(SolverRunnerError::SendFailed);
    }
    return Ok(());
}
```

```
//| id: sorr_try-join
pub fn try_join(&mut self) -> Result<SatResult,
SolverRunnerError<S::Error>> {
    let handle = match self.join_handle.as_ref() {
        Some(handle) => {
            if handle.is_finished() {
                self.join_handle.take().unwrap()
            }
            else {
                return Err(SolverRunnerError::NotFinished)
```

```
            }
        },
        None => return Err(SolverRunnerError::AlreadyJoined),
    };

    match handle.join() {
        Ok(Ok(ret)) => return Ok(ret),
        Ok(Err(err)) => return
Err(SolverRunnerError::SolverError(err)),
        Err(_) => return Err(SolverRunnerError::JoinPanicked)
    }
}
```

```
//| file: rust/viska-sat/src/solver_runner.rs
<<sor_modules>>
<<sor_solver-runner-event-handler>>
<<sor_error>>
<<sor_runner>>
```

第 2 章
アルゴリズム

2.1 全探索

2.1.1 概要

「式を充足させる割り当てが存在するか？」という問いに確実に答えるためには、ありえる割り当てを全て試せばよい。完全な割り当ては各変数に対して真か偽のどちらかを対応付けたものなので、n 変数の完全な割り当ては 2^n 通りある。

2.1.2 実装

今回は再帰関数を使って実装する。

```
//|  id: brf_brute-force
fn brute_force(&mut self, idx: usize, assign: &mut
Assignment) -> Result<SatResult, H::Error> {
    <<brf_base-case>>
    <<brf_recursive-step>>
}
```

`assign` はこれまでの割り当てを表す。 `brute_force()` は `assign`

を含む完全な割り当て[1] であって、式を充足するものはあるかどうかを返り値として返す。簡略化のために、`idx` はまだ決定されていない変数の最小の id を保持して、`assign` は `idx` より小さいの id の変数全てが割り当てられているようにする。

2.1.2.1　ベースケース

`assign` が完全なら、その割り当てで式を評価した結果を `SatResult` で返す。

```
//|id: brf_base-case
if assign.is_full() {
    let sat_state = self.cnf.eval(assign);
    self.handler.handle_event(BruteForceSolverEvent::Eval
{ result: sat_state.clone() })?;
    match sat_state {
        CnfState::Satisfied => return
Ok(SatResult::Sat(assign.clone())),
        CnfState::Unsatisfied => return Ok(SatResult::Unsat),
        CnfState::Unresolved => panic!("full assignment
cannot be unresolved")
    };
}
```

2.1.2.2　再帰ステップ

`assign` に `idx` 番目の変数の割り当てを追加する。真に割り当てた場合に充足すれば `Sat` を返す。 充足しなければ、偽に割り当てた場合を調べる。 充足すれ `Sat` を返し、そうでなければ `Unsat` を返す。

---

[1] 割り当て A, B について、B で割り当てらている変数全てについて、 B で割り当てられていた値で A に割り当てられていることを A が B を含むという。

```
//| id: brf_recursive-step
let mut ret = SatResult::Unsat;
for choice in [true, false] {
    self.handler.handle_event(BruteForceSolverEvent::Decide
{ idx, assign: choice })?;
    assign.values[idx] = Some(choice);
    let result = self.brute_force(idx + 1, assign)?;

self.handler.handle_event(BruteForceSolverEvent::Backtrack
{ idx })?;
    match result {
        sat @ SatResult::Sat(_) => {
            ret = sat;
            break;
        }
        SatResult::Unsat => {}
    }
}
assign.values[idx] = None;
return Ok(ret);
```

```
//| file: rust/viska-sat/src/brute_force.rs
use crate::{assignment::Assignment, cnf::{Cnf, CnfState},
event_handler::EventHandler, solver::{SatResult, Solver}};

#[derive(Debug)]
pub enum BruteForceSolverEvent {
    Decide {idx: usize, assign: bool},
    Eval {result: CnfState},
    Backtrack {idx: usize},
    Finish {result: SatResult}
}
```

```
pub struct BruteForceSolver<H>
{
    pub cnf: Cnf,
    pub handler: H
}

impl<H> BruteForceSolver<H>
where
    H: EventHandler<Event = BruteForceSolverEvent>
{
    <<brf_brute-force>>
}

impl<H> Solver for BruteForceSolver<H>
where
    H: EventHandler<Event = BruteForceSolverEvent>
{
    type Event = BruteForceSolverEvent;
    type Handler = H;
    type Error = H::Error;

    fn solve(&mut self) -> Result<SatResult, Self::Error> {
        let result = self.brute_force(0, &mut Assignment
{ values: vec![None; self.cnf.num_vars]})?;

self.handler.handle_event(BruteForceSolverEvent::Finish
{ result: result.clone() })?;
        Ok(result)
    }
}
```

2.2 DPLL

2.2.1 概要

全探索は確実に問題を解くことができる反面、$O(2^n)$ の計算量では n が大きくなったときに実行が現実的な時間で終わらない。全探索の探索空間を削減することで、多少の改善を試みる。

たとえば以下の CNF について考える：

$$(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_2 \vee \neg x_3)$$

ここで、$\{x_1 \mapsto \text{true}\}$ と割り当てると、節 $x_1 \vee x_2 \vee x_3$ は充足し、 節 $x_1 \vee \neg x_2 \vee \neg x_3$ は同様に充足する。したがって、問題の式は $\neg x_1 \vee \neg x_2$ に簡約される。

ここで、$\{x_1 \mapsto \text{true}, x_2 \mapsto \text{true}\}$ と割り当てを追加したとしよう。 すると、$\neg x_1 \vee \neg x_2$ は**矛盾**（conflict）となってしまい、x_3 をどのように割り当てても充足不能のままとなる。

実際には、$\neg x_1 \vee \neg x_2$ において、節内に登場するリテラルが 1 つ（$\neg x_2$）を除いて全て割り当てられていて、その全てが割り当てによって充足していないという条件から、 自動的に x_2 は $\neg x_2$ を充足するような割り当て、つまり $\{x_2 \mapsto \text{false}\}$ を追加することができる。この条件をみたす節を**単位節**と言い、その単位節における未割り当てのリテラルを充足するように割り当てを追加することができる。これを**単位伝播**という。

全探索と単位伝播を組み合わせることで、明らかに充足不能な枝の探索を削減できる。新たな割り当てを追加するたびに、単位節がなくなるまで単位伝播をすることでこれを実現した。また、完全な割り当てでなくとも 1 つの節が充足不能になってしまえば全体として充足不

能となるので、この場合もすぐに探索を終了するようにした。今回は純リテラル除去については触れないことにする。

2.2.2　実装

全探索ソルバでは `idx` を使ってどこまで割り当てたかを管理していた。しかし、単位伝播を採用したことで id が小さい順に割り当てられていく保証がなくなったので、割り当ての中で未割り当てである変数の最も小さい id を取得する関数 `pick_unassigned_var()` を用意することで一旦対処する。

```
//| id: dpll_pick-unassigned-var
fn pick_unassigned_var(&self, assign: &Assignment) ->
Option<usize> {
    for i in 0..assign.values.len() {
        if assign.values[i].is_none() {
            return Some(i);
        }
    }
    return None;
}
```

ソルバの構造は以下のようになっている：

1. 単位伝播を進展が生まれなくなるまで実行する。このとき伝播した変数の id をスタックに入れておく。充足不能となるとすぐに中断する。
2. 割り当てが完全なとき、式を割り当てで評価してその結果を返す。
3. 未割り当ての変数を選んで真か偽にそれぞれ割り当てて再帰する。

```
//| id: dpll_dpll
fn dpll(&mut self, assign: &mut Assignment) ->
Result<SatResult, H::Error> {
    let mut ret = SatResult::Unsat;
    <<dpll_unit-propagation-and-conflict>>
    <<dpll_eval-with-assignment>>
    <<dpll_decide>>
    <<dpll_return>>
}
```

単位伝播を単位節がなくなるもしくは矛盾が発生するまで繰り返す関数を用意する。 返り値に現在の割り当てで式を評価した結果を返す。

```
//| id: dpll_repeat-unit-propagation
fn repeat_unit_propagate(
    &mut self,
    assign: &mut Assignment,
    propagated_vars: &mut Vec<usize>
) -> Result<CnfState, H::Error> {
    'outer: loop {
        for (clause_id, clause) in
self.cnf.clauses.iter().enumerate() {
            match clause.eval(assign) {
                ClauseState::Unit(lit) => {
                    let idx = lit.var_id;
                    let val = !lit.negated;
                    self.handler.handle_event(
                        DpllSolverEvent::Propagated { idx,
assign: val, reason: clause_id }
                    )?;
                    assign.values[idx] = Some(val);
                    propagated_vars.push(idx);
```

```
                    continue 'outer;
                }
                ClauseState::Unsatisfied => break 'outer,
                _ => {}
            }
        }
        break;
    }
    Ok(self.cnf.eval(assign))
}
```

`propagated_vars` に単位伝播された変数を保管する。`unit_propagate()` で矛盾が発生したなら、充足不能と返す。

```
//| id: dpll_unit-propagation-and-conflict
let mut propagated_vars = vec![];
if let CnfState::Unsatisfied =
self.repeat_unit_propagate(assign, &mut propagated_vars)? {
    self.handler.handle_event(DpllSolverEvent::Eval { result:
CnfState::Unsatisfied })?;
    ret = SatResult::Unsat;
}
```

割り当てが完全なとき、式を評価してその結果を返す。

```
//| id: dpll_eval-with-assignment
else if assign.is_full() {
    let sat_state = self.cnf.eval(assign);
    self.handler.handle_event(DpllSolverEvent::Eval { result:
sat_state.clone() })?;
    match sat_state {
        CnfState::Satisfied => return
Ok(SatResult::Sat(assign.clone())),
        CnfState::Unsatisfied => return Ok(SatResult::Unsat),
```

```
        CnfState::Unresolved => panic!("full assignment
cannot be unresolved")
    };
}
```

式が未解決のときは、全探索のときと同じように変数を決定する。`idx` の扱いが全探索の場合とは異なっているが、だいたい同じことをしている。

```
//| id: dpll_decide
else {
    let idx =
self.pick_unassigned_var(assign).expect("branching called
with fully assigned assignment");
    for choice in [true, false] {
        self.handler.handle_event(DpllSolverEvent::Decide
{ idx, assign: choice })?;
        assign.values[idx] = Some(choice);
        let result = self.dpll(assign)?;
        self.handler.handle_event(DpllSolverEvent::Backtrack
{ idx })?;
        match result {
            sat @ SatResult::Sat(_) => {
                ret = sat;
                break;
            }
            SatResult::Unsat => {}
        }
    }
    assign.values[idx] = None;
}
```

最後に結果を返す。

```
//| id: dpll_return
while let Some(var_id) = propagated_vars.pop() {
    assign.values[var_id] = None;
    self.handler.handle_event(DpllSolverEvent::Backtrack
{ idx: var_id })?;
}
return Ok(ret);
```

```
//| file: rust/viska-sat/src/dpll.rs
use crate::{assignment::Assignment, clause::ClauseState,
cnf::{Cnf, CnfState}, event_handler::EventHandler, solver::
{SatResult, Solver}};

#[derive(Debug)]
pub enum DpllSolverEvent {
    Decide {idx: usize, assign: bool},
    Propagated {idx: usize, assign: bool, reason: usize},
    Eval {result: CnfState},
    Backtrack {idx: usize},
    Finish {result: SatResult}
}

pub struct DpllSolver<H>
{
    pub cnf: Cnf,
    pub handler: H
}

impl<H> DpllSolver<H>
where
    H: EventHandler<Event = DpllSolverEvent>
{
    <<dpll_pick-unassigned-var>>
```

```
    <<dpll_repeat-unit-propagation>>

    <<dpll_dpll>>
}

impl<H> Solver for DpllSolver<H>
where
    H: EventHandler<Event = DpllSolverEvent>
{
    type Event = DpllSolverEvent;
    type Handler = H;
    type Error = H::Error;

    fn solve(&mut self) -> Result<SatResult, Self::Error> {
        let result = self.dpll(&mut Assignment { values: vec!
[None; self.cnf.num_vars]})?;
        self.handler.handle_event(DpllSolverEvent::Finish
{ result: result.clone() })?;
        Ok(SatResult::Unsat)
    }
}
```

2.3 CDCL の基礎

2.3.1 概要

単位伝播によって DPLL では探索空間を削減していたが、 ここではさらなる効率化を行う。

以下の CNF を考えてみよう：

$$(\neg x_1 \vee x_2) \wedge (\neg x_3 \vee x_5) \wedge (\neg x_4 \vee x_5) \wedge (\neg x_6 \vee \neg x_7)$$
$$\wedge (\neg x_1 \vee \neg x_5 \vee x_6) \wedge (\neg x_2 \vee \neg x_5 \vee x_7)$$

これを DPLL で解いてみると次のようなステップになる：

1. $\{x_1 \mapsto \text{true}\}$ と決定する。
2. 単位伝播により $\{x_2 \mapsto \text{true}\}$。
3. $\{x_3 \mapsto \text{true}\}$ と決定する。
4. 単位伝播により $\{x_5 \mapsto \text{true}, x_6 \mapsto \text{true}, x_7 \mapsto \text{true}\}$。
5. $\neg x_6 \vee \neg x_7$ が非充足となり矛盾。 x_3, x_5, x_6, x_7 の割り当てを削除する。
6. $\{x_3 \mapsto \text{false}\}$ と決定する。
7. $\{x_4 \mapsto \text{true}\}$ と決定する。
8. 単位伝播により $\{x_5 \mapsto \text{true}, x_6 \mapsto \text{true}, x_7 \mapsto \text{true}\}$。
9. $\neg x_6 \vee \neg x_7$ が非充足となり矛盾。 x_4, x_5, x_6, x_7 の割り当てを削除する。
10. （以下省略）

ここで 4, 5 と 8, 9 で行なっている作業はほとんど同じである。 4, 5 の経験を活かして 8, 9 の探索を削減できることが望ましい。矛盾の原因を学習して同じ間違いを繰り返さないようにする仕組みを備えたものが CDCL ソルバである。

CDCL のコンセプトを説明する前に基本的な用語を定義する：

決定レベル　各変数における割り当てがどの決定に結び付いているかを表す。たとえば決定レベル 1 の変数は、1 回目の決定に由来するということを表す。

バックトラック　指定された決定レベルより大きい決定レベルの変数の割り当てを削除すること。

CDCL の**学習**の基本原理として**導出原理**という法則がある。 リテラル $a_1, a_2, ..., a_m, b_1, b_2, ..., b_n, \alpha$ について、

$$\left(a_1 \vee a_2 \vee ... \vee a_m \vee \alpha\right) \wedge \left(b_1 \vee b_2 \vee ... \vee b_n \vee \neg\alpha\right)$$

から

$$a_1 \vee a_2 \vee ... \vee a_m \vee b_1 \vee b_2 \vee ... \vee b_n$$

が導かれるという法則のことである。

ここで、矛盾の根本的な原因を考察してみる。決定レベルが d であるような決定をして、その後の単位伝播によって、節 c が矛盾することが分かった。 ここで、決定レベルが $d-1$ の時点では矛盾する節はないので、節 c には決定レベル d で決定されたリテラルとそれに由来する伝播されたリテラルが含まれている。節 c に含まれる決定レベル d で伝播されたリテラルを α とおく。 すると、リテラル $a_1, a_2, ..., a_m$ を用いて節 c は次のように書ける：

$$c = a_1 \vee a_2 \vee ... \vee a_m \vee \neg\alpha$$

ここで、決定レベル d において α が伝播されていることから、その伝播の理由となる単位節が存在する。その単位節を u として、リテラル $b_1, b_2, ..., b_n$ を用いて節 u は次のように書ける：

$$u = b_1 \vee b_2 \vee ... \vee b_n \vee \alpha$$

したがって、導出原理により以下の節 l が導かれる：

$$l = a_1 \vee a_2 \vee ... \vee a_m \vee b_1 \vee b_2 \vee ... \vee b_n$$

ここで、この節を式に加えても充足可能性は変化しないので、この得られた節を**学習節**として加えても問題ない。この一連の操作によって学習節を生成することを**学習**という。

この得られた学習節の意味を考えてみよう。もともと、矛盾の原因はリテラル $a_1, a_2, ..., a_m$ と $\neg\alpha$ が全て充足しないことであった。そもそも $\neg\alpha$ が充足しない原因は α が充足することであって、それは単位節 u による伝播が原因であった。それは節 u が単位節になること、つまり $b_1, b_2, ..., b_n$ が全て充足しないことである。だから結局 $a_1, a_2, ..., a_m$ と $b_1, b_2, ..., b_n$ が全て充足しないことが矛盾の原因である。よって、学習節として、これらのリテラルのどれかが充足するという条件を表す節 l が得られる。こうして、α を使わずに矛盾の原因を表現することができた。

ここで注目したいのは、節 c に含まれる決定レベル d で伝播されたリテラルであれば同じ操作ができて、そのリテラルを使わない形で矛盾の原因を表現することができるということである。この操作をできる限り繰り返すことで、決定レベルが d であるようなリテラルがただ 1 つのみ含まれるような学習節 l' を必ず作ることができる。そのリテラルを β とする。すると、現在の割り当てでは l' 内のリテラルは全て充足しないことを踏まえれば、l' のリテラルの決定レベルの中で 2 番目に大きい決定レベル、つまり決定レベルが d でないもののうち最も大きい決定レベルにバックトラックすると 節 l' はただちに単位節となる。したがって、すぐに β が伝播される。矛盾の原因を解析することで大幅に探索空間が削減されることが分かる。この解析を**矛盾の解析**などと呼んだりする。

この一連の操作は**含意グラフ**を用いることで視覚的に捉えることができる。ただし、この部分はビジュアライザのドキュメントの方に譲ろうと思う。

経験的に、矛盾節に含まれる決定レベル d のリテラルであって、最も直近に割り当てられたものから順番に導出原理を適用させることで、得られる学習節が短く効果的になることが知られているそう。実装が簡便であることから、今回はこの方針を取る。

実際に具体例を通してどのように CDCL の学習が効果的に働くかを見てみよう。 以下のように節に名前を付けて、その連言を考える。

$$c_1 = \neg x_1 \vee x_2, \quad c_2 = \neg x_3 \vee x_5,$$
$$c_3 = \neg x_4 \vee x_5, \quad c_4 = \neg x_6 \vee \neg x_7,$$
$$c_5 = \neg x_1 \vee \neg x_5 \vee x_6, \quad c_6 = \neg x_2 \vee \neg x_5 \vee x_7$$

- 決定（レベル 1）
  - $\{x_1 \mapsto \text{true}\}$ と決定する。
  - 単位伝播により $\{x_2 \mapsto \text{true}\}$。
- 決定（レベル 2）
  - $\{x_3 \mapsto \text{true}\}$ と決定する。
  - 単位伝播により $\{x_5 \mapsto \text{true}, x_6 \mapsto \text{true}, x_7 \mapsto \text{true}\}$。
  - c_4 が非充足となり矛盾。
- 矛盾の解析
  - 矛盾節は $\neg x_6 \vee \neg x_7$
  - x_7 の伝播理由は c_6。 ここで導出原理により、$\neg x_2 \vee \neg x_5 \vee \neg x_6$。
  - x_6 の伝播理由は c_5。 ここで導出原理により、$\neg x_1 \vee \neg x_2 \vee \neg x_5$
  - 決定レベルが 2 であるリテラルが $\neg x_5$ になったので終了。これを学習節 $c_7 = \neg x_1 \vee \neg x_2 \vee \neg x_5$ とする。
- バックトラック
  - c_7 において 2 番目に大きい決定レベルは 1 なので、決定レベル 1 にバックトラックする。
  - 割り当ては $\{x_1 \mapsto \text{true}, x_2 \mapsto \text{true}\}$。
  - 単位伝播により、$\{x_5 \mapsto \text{false}, x_6 \mapsto \text{false}, x_7 \mapsto \text{false}\}$

・（以下省略）

これを DPLL と比較すれば、学習節によって、直ちに $\{x_5 \mapsto \text{false}\}$ が結論付けらていて、 無駄な探索が減っていることが分かる。

第 3 章

Godot

3.1　エントリーポイント

godot-rust API の主要部分を読み込む。

```
//| id: grl_godot-rust-api
use godot::prelude::*;
```

ファイル分割して記述したモジュールを読み込む。

```
//| id: grl_modules
mod tests;
```

空の構造体 `ViskaSATExtension` を作って、GDExtension 用のエントリーポイントにする。Godot とやりとりをする部分だから `unsafe` になっている。

```
//| id: grl_gdextension-entry-point
struct ViskaSATExtension;

#[gdextension]
unsafe impl ExtensionLibrary for ViskaSATExtension {}
```

```
//| file: rust/godot-rust/src/lib.rs
<<grl_godot-rust-api>>
<<grl_modules>>

<<grl_gdextension-entry-point>>
```

第 4 章
テスト

4.1　テスト用モジュール

　様々な内容の動作確認のために書いたコードを雑多にまとめておく。

```
//| file: rust/godot-rust/src/tests.rs
pub mod thread_channel_communication;
pub mod solver_trait;
pub mod solver_communicator;
pub mod solver_runner;
```

4.2　スレッド間のチャネルの通信

　`Solver` と `SolverRunner` 間の通信の中核をなすチャネルについてテストしてみる。

　スレッドとチャネルのモジュールを読み込む。

```
//| id: tcc_modules
use std::thread;
use std::sync::mpsc;
use std::time::Duration;
```

4.2.1 `init`

フィールドの初期化をする。

```
//| id: tcc_init
fn init(base: Base<Control>) -> Self {
    Self {
        event_rx: None,
        ctrl_tx: None,
        is_pause: true,
        base
    }
}
```

4.2.2 `ready`

チャネルを立ててフィールドに代入したり、その他変数を用意する。

```
//| id: tcc_init_vars
let (event_tx, event_rx) = mpsc::channel::<u64>();
let (ctrl_tx, ctrl_rx) = mpsc::channel::<bool>();
self.event_rx = Some(event_rx);
self.ctrl_tx = Some(ctrl_tx);

let mut is_pause = self.is_pause;
```

そして、1 秒ごとに数字をカウントアップするスレッドを立てる。

```
//| id: tcc_thread
thread::spawn(move || {
    for val in 0..=100 {
        <<tcc_pause-handle>>
        event_tx.send(val).unwrap();
        thread::sleep(Duration::from_secs(1));
```

```
    }
});
```

ただし、一時停止のメッセージを受け取ったら再開のメッセージを受け取るまで停止する。

```
//| id: tcc_pause-handle
while let Ok(received) = ctrl_rx.try_recv() {
    is_pause = received;
}
while is_pause {
    let mut pause_flag = ctrl_rx.recv().unwrap();
    while let Ok(received) = ctrl_rx.try_recv() {
        pause_flag = received;
    }
    is_pause = pause_flag;
}
```

```
//| id: tcc_ready
fn ready(&mut self) {
    <<tcc_init_vars>>
    <<tcc_thread>>
}
```

4.2.3 `process`

そもそもチャネルが作られているか確認する。

```
//| id: tcc_check-channel
let (event_rx, ctrl_tx) = match (&self.event_rx,
&self.ctrl_tx) {
    (Some(rx), Some(tx)) => (rx, tx),
```

```
    _ => return,
};
```

もしデータがあるなら受け取る。

```
//| id: tcc_receive
if let Ok(received) = event_rx.try_recv() {
    godot_print!("{}", received);
}
```

決定ボタンが押されたらカウントアップの一時停止・再開をする。

```
//| id: tcc_pause-stop
let input = Input::singleton();
if input.is_action_just_pressed("ui_accept") {
    self.is_pause = !self.is_pause;
    godot_print!("is_pause: {}", self.is_pause);
    ctrl_tx.send(self.is_pause).unwrap();
}
```

```
//| id: tcc_process
fn process(&mut self, _delta: f64) {
    <<tcc_check-channel>>
    <<tcc_receive>>
    <<tcc_pause-stop>>
}
```

```
//| file: rust/godot-rust/src/tests/
thread_channel_communication.rs
use godot::prelude::*;
use godot::classes::{Control, IControl};
<<tcc_modules>>
```

```
#[derive(GodotClass)]
#[class(base=Control)]
struct ThreadChannelCommunication {
    event_rx: Option<mpsc::Receiver<u64>>,
    ctrl_tx: Option<mpsc::Sender<bool>>,
    is_pause: bool,
    base: Base<Control>
}

#[godot_api]
impl IControl for ThreadChannelCommunication {
    <<tcc_init>>
    <<tcc_ready>>
    <<tcc_process>>
}
```

4.3 Solver トレイト

1.3 節で定義した Solver トレイトをテストしてみる。4.2 節の一部を Solver トレイトを実装した構造体に置き換えて同様に動作することを確認する。

```
//| id: sot_modules
use viska_sat::{solver::{Solver, SatResult},
event_handler::EventHandler};
use std::sync::mpsc::{channel, Sender, Receiver,
TryRecvError};

use std::time::Duration;
use std::thread;
```

また、適当なエラーの型を作っておく。

```
//| id: sot_error-type
#[derive(Debug)]
struct TestError;
```

4.3.1 `TestHandler`

イベントハンドラーを作る。 4.2 節のそれとほぼ同じような内容にする。

```
//| id: sotth-decl
struct TestHandler {
    event_tx: Sender<u64>,
    ctrl_rx: Receiver<bool>,
    is_pause: bool
}
```

関連型を設定する。 `Event` は今の進捗、つまりカウントアップした数字を表せるように `u64` にした。 また、 `Error` は適当に作った `TestError` にしておく。

```
//| id: sotth_associated-types
type Event = u64;
type Error = TestError;
```

最新の制御メッセージを取る。

```
//| id: sotth_try-recv-latest
loop {
    match self.ctrl_rx.try_recv() {
        Ok(received) => self.is_pause = received,
        Err(TryRecvError::Empty) => break,
        Err(TryRecvError::Disconnected) => return
Err(TestError),
```

```
    }
}
```

停止中なら再開の制御メッセージが届くまで待機する。

```
//| id: sotth_recv-latest
while self.is_pause {
    self.is_pause = match self.ctrl_rx.recv() {
        Ok(val) => val,
        Err(_) => return Err(TestError),
    };

    <<sotth_try-recv-latest>>
}
```

イベントを送信する。

```
//| id: sotth_send-event
if self.event_tx.send(event).is_err() {
    return Err(TestError);
}
```

```
//| id: sot_test-handler
<<sotth-decl>>

impl EventHandler for TestHandler {
    <<sotth_associated-types>>

    fn handle_event(&mut self, event: Self::Event) ->
Result<(), Self::Error> {
        <<sotth_try-recv-latest>>
        <<sotth_recv-latest>>
        <<sotth_send-event>>
```

```
        Ok(())
    }
}
```

4.3.2 `DummySolver`

実際にソルバとしては機能しない、ただ数字をカウントアップするだけのダミー `DummySolver` を作る。

```
//| id: sotds_decl
struct DummySolver<H> {
    handler: H,
}
```

関連型を設定する。これは `TestHandler` と同じ。

```
//| id: sotds_associated-types
type Event = u64;
type Error = TestError;
type Handler = H;
```

ソルバの中身を定義する。

```
//| id: sotds_solve
fn solve(&mut self) -> Result<viska_sat::solver::SatResult,
Self::Error> {
    for val in 0..=100 {
        if self.handler.handle_event(val).is_err() {
            return Err(TestError);
        }
        thread::sleep(Duration::from_secs(1));
    }
```

```
    Ok(SatResult::Unsat)
}
```

```
//| id: sot_dummy-solver
<<sotds_decl>>
impl<H> Solver for DummySolver<H>
where
    H: EventHandler<Event = u64, Error = TestError>
{
    <<sotds_associated-types>>
    <<sotds_solve>>
}
```

あとはこれを動かすだけ。 重要な部分だけピックアップする。

4.3.3 `ready`

チャネルを立てて、`EventHandler` に渡す。

```
//| id: sotr_start-channel
let (event_tx, event_rx) = channel::<u64>();
let (ctrl_tx, ctrl_rx) = channel::<bool>();
self.event_rx = Some(event_rx);
self.ctrl_tx = Some(ctrl_tx);
let handler = TestHandler {
    event_tx,
    ctrl_rx,
    is_pause: self.is_pause,
};
```

`DummySolver` を作って解き始める。

```
//| id: sotr_start-solving
let mut solver = DummySolver {
```

```
    handler
};
thread::spawn(move || {
    solver.solve().unwrap();
});
```

```
//| id: sot_ready
fn ready(&mut self) {
    <<sotr_start-channel>>
    <<sotr_start-solving>>
}
```

4.3.4 `process`

`process` は 4.2 節の実装をそのまま使用する。

```
//| file: rust/godot-rust/src/tests/solver_trait.rs
use godot::prelude::*;
use godot::classes::{Control, IControl};
<<sot_modules>>
<<sot_error-type>>
<<sot_test-handler>>
<<sot_dummy-solver>>


#[derive(GodotClass)]
#[class(base=Control)]
struct SolverTrait {
    event_rx: Option<Receiver<u64>>,
    ctrl_tx: Option<Sender<bool>>,
    is_pause: bool,
    base: Base<Control>
}
```

```
#[godot_api]
impl IControl for SolverTrait {
    fn init(base: Base<Control>) -> Self {
        Self {
            event_rx: None,
            ctrl_tx: None,
            is_pause: true,
            base
        }
    }

    <<sot_ready>>
    <<tcc_process>>
}
```

4.4 SolverCommunicator

SolverCommunicator のテストをする。 引き続き同じ内容を実装しながら一部を差し替えていく。 4.3 節のほとんどと内容が共通するので変更点だけピックアップする。

新たに SolverCommunicator を読み込む。

```
//| id: soc_modules
use viska_sat::{solver::{Solver, SatResult},
event_handler::EventHandler, solver_communicator::
{SolverCommunicator, SolverControl,
SolverCommunicatorError}};
use std::sync::mpsc::{channel, Sender, Receiver};
```

```
use std::time::Duration;
use std::thread;
```

4.4.1 `TestHandler`

`SolverCommunicator` を使って `TestHandler` を書き直す。`Sender`, `Receiver` を直接持つのではなく、かわりに `SolverCommunicator` を使うようにした。

```
//| id: socth-decl
struct TestHandler {
    com: SolverCommunicator<u64>,
    is_pause: bool
}
```

最新の制御があれば受けとって、停止かどうかを `is_pause` に入れる。

```
//| id: socth_try-recv-latest
match self.com.try_recv_latest_control() {
    Ok(Some(receive)) => {
        self.is_pause = receive == SolverControl::Pause;
    }
    Err(err) => return Err(err),
    _ => {}
};
```

もし停止する必要があるなら、再開の制御を受けとるまでループする。

```
//| id: socth_pause-loop
while self.is_pause {
    match self.com.recv_latest_control() {
```

```
        Ok(receive) => {
            self.is_pause = receive == SolverControl::Pause;
        }
        Err(err) => return Err(err),
    }
}
```

最後にイベントを送信する。

```
//| id: socth_send-event
if let Err(err) = self.com.send_event(event) {
    return Err(err);
}
```

```
//| id: soc_test-handler
<<socth-decl>>

impl EventHandler for TestHandler {
    type Event = u64;
    type Error = SolverCommunicatorError;

    fn handle_event(&mut self, event: Self::Event) ->
Result<(), Self::Error> {
        <<socth_try-recv-latest>>
        <<socth_pause-loop>>
        <<socth_send-event>>
        Ok(())
    }
}
```

4.4.2 DummySolver

型を現行に追従するように修正した。

```
//| id: socds_associated-types
type Event = u64;
type Error = SolverCommunicatorError;
type Handler = H;
```

ソルバの中身はだいたいそのまま。

```
//| id: socds_solve
fn solve(&mut self) -> Result<viska_sat::solver::SatResult,
Self::Error> {
    for val in 0..=100 {
        if let Err(err) = self.handler.handle_event(val) {
            return Err(err);
        }
        thread::sleep(Duration::from_secs(1));
    }
    Ok(SatResult::Unsat)
}
```

```
//| id: soc_dummy-solver
<<sotds_decl>>
impl<H> Solver for DummySolver<H>
where
    H: EventHandler<Error = SolverCommunicatorError, Event =
u64>
{
    <<socds_associated-types>>
    <<socds_solve>>
}
```

4.4.3 `ready`

フィールドと型を調整した。

```
//| id: socr_start-channel
let (event_tx, event_rx) = channel::<u64>();
let (ctrl_tx, ctrl_rx) = channel::<SolverControl>();
self.event_rx = Some(event_rx);
self.ctrl_tx = Some(ctrl_tx);
let handler = TestHandler {
    com: SolverCommunicator::new(event_tx, ctrl_rx),
    is_pause: self.is_pause,
};
```

```
//| id: soc_ready
fn ready(&mut self) {
    <<socr_start-channel>>
    <<sotr_start-solving>>
}
```

4.4.4 process

`SolverControl` の表現に直した。

```
//| id: soc_pause-stop
let input = Input::singleton();
if input.is_action_just_pressed("ui_accept") {
    self.is_pause = !self.is_pause;
    godot_print!("is_pause: {}", self.is_pause);
    ctrl_tx.send(if self.is_pause {SolverControl::Pause} else
{SolverControl::Resume} ).unwrap();
}
```

```
//| id: soc_process
fn process(&mut self, _delta: f64) {
    <<tcc_check-channel>>
```

```
    <<tcc_receive>>
    <<soc_pause-stop>>
}
```

```
//| file: rust/godot-rust/src/tests/solver_communicator.rs
use godot::prelude::*;
use godot::classes::{Control, IControl};
<<soc_modules>>
<<soc_test-handler>>
<<soc_dummy-solver>>


#[derive(GodotClass)]
#[class(base=Control)]
struct SolverCommunicatorTest {
    event_rx: Option<Receiver<u64>>,
    ctrl_tx: Option<Sender<SolverControl>>,
    is_pause: bool,
    base: Base<Control>
}

#[godot_api]
impl IControl for SolverCommunicatorTest {
    fn init(base: Base<Control>) -> Self {
        Self {
            event_rx: None,
            ctrl_tx: None,
            is_pause: true,
            base
        }
    }

    <<soc_ready>>
```

```
    <<soc_process>>
}
```

4.5 SolverRunner

例によって同じ内容を SolverRunner を使って書き換えてテストする。

モジュールを調整。

```
//| id: sort_modules
use viska_sat::{solver::{Solver, SatResult},
event_handler::EventHandler, solver_communicator::
{SolverControl, SolverCommunicatorError}, solver_runner::
{SolverRunner, SolverRunnerEventHandler}};

use std::time::Duration;
use std::thread;
```

```
//| id: sort_ready
fn ready(&mut self) {
    self.runner = Some(SolverRunner::start_solver(|handler|
DummySolver{handler}));
}
```

```
//| id: sort_process
fn process(&mut self, _delta: f64) {
    let runner = match &self.runner {
        Some(r) => r,
        None => return
    };
```

```
    if let Ok(Some(received)) = runner.try_recv_event() {
        godot_print!("{}", received);
    }

    let input = Input::singleton();
    if input.is_action_just_pressed("ui_accept") {
        self.is_pause = !self.is_pause;
        godot_print!("is_pause: {}", self.is_pause);
        let con = if self.is_pause {SolverControl::Pause}
else {SolverControl::Resume};
        runner.send_control(con).unwrap();
    }
}
```

```
//| file: rust/godot-rust/src/tests/solver_runner.rs
use godot::prelude::*;
use godot::classes::{Control, IControl};
<<sort_modules>>
<<soc_dummy-solver>>

type Runner =
SolverRunner<DummySolver<SolverRunnerEventHandler<u64>>>;

#[derive(GodotClass)]
#[class(base=Control)]
struct SolverRunnerTest {
    runner: Option<Runner>,
    is_pause: bool,
    base: Base<Control>
}

#[godot_api]
impl IControl for SolverRunnerTest {
```

```
    fn init(base: Base<Control>) -> Self {
        Self {
            runner: None,
            is_pause: false,
            base
        }
    }

    <<sort_ready>>
    <<sort_process>>
}
```

4.6 ソルバのテスト

4.6.1 共通部

これから様々なソルバを作っていくことになるので、共通してテストできるようにする。以下のテストを用意した：

`solve_with_logging()` CNF を 1 つだけ解く。このときはログを出すようにする。

`solve_many_small()` 小さな CNF をたくさん解く。

`solve_many_large()` 大きな CNF をたくさん解く。

そのために、CNF・ハンドラ・ソルバ・クロージャを引数に取って問題を解く関数 `run_solver()` を定義する。

```
//| id: vst_run-solver
fn run_solver<S, H, F>(cnf: Cnf, handler: H, make_solver: F)
-> (Result<SatResult, S::Error>, Duration)
where
```

```
    S: Solver,
    H: EventHandler,
    F: FnOnce(Cnf, H) -> S
{
    let mut solver = make_solver(cnf, handler);
    let start = Instant::now();
    let result = solver.solve();
    let elapsed = start.elapsed();
    (result, elapsed)
}
```

`solve_with_logging()` では用意されたいくつかの CNF を選んで解くことができる。

1. 2.2 節で出てきた CNF。

$$(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_2 \vee \neg x_3)$$

```
//| id: vst_sample-cnf
let sample_cnfs = vec![
    Cnf {
        num_vars: 3,
        clauses: vec![
            Clause { lits: vec![
                Lit { var_id: 0, negated: false },
                Lit { var_id: 1, negated: false },
                Lit { var_id: 2, negated: false },
            ], meta: () },
            Clause { lits: vec![
                Lit { var_id: 0, negated: true },
                Lit { var_id: 1, negated: true },
            ], meta: () },
            Clause { lits: vec![
```

```
                Lit { var_id: 0, negated: false },
                Lit { var_id: 1, negated: true },
                Lit { var_id: 2, negated: true },
            ], meta: () },
        ],
    },
    Cnf {
        num_vars: 7,
        clauses: vec![
            Clause { lits: vec![
                Lit { var_id: 0, negated: true },
                Lit { var_id: 1, negated: false },
            ], meta: () },
            Clause { lits: vec![
                Lit { var_id: 2, negated: true },
                Lit { var_id: 4, negated: false },
            ], meta: () },
            Clause { lits: vec![
                Lit { var_id: 3, negated: true },
                Lit { var_id: 4, negated: false },
            ], meta: () },
            Clause { lits: vec![
                Lit { var_id: 5, negated: true },
                Lit { var_id: 6, negated: true },
            ], meta: () },
            Clause { lits: vec![
                Lit { var_id: 0, negated: true },
                Lit { var_id: 4, negated: true },
                Lit { var_id: 5, negated: false },
            ], meta: () },
            Clause { lits: vec![
                Lit { var_id: 1, negated: true },
                Lit { var_id: 4, negated: true },
```

```
                Lit { var_id: 6, negated: false },
            ], meta: () },
        ],
    }
];
```

これを解かせるときに、ログを出すようにしたいので、ログを出すハンドラを定義する。

```
//| id: vst_logger-handler
pub struct LoggerHandler<E: Debug> {
    _marker: PhantomData<E>
}

impl<E: Debug> LoggerHandler<E> {
    fn new() -> Self {
        Self {
            _marker: PhantomData
        }
    }
}

impl<E: Debug> EventHandler for LoggerHandler<E>
{
    type Event = E;
    type Error = ();

    fn handle_event(&mut self, event: Self::Event) ->
Result<(), Self::Error> {
        println!("{:?}", event);
        Ok(())
    }
}
```

そして、このハンドラを渡してソルバのインスタンスを作り、走らせる。

```
//| id: vst_solve-with-logging
pub fn solve_with_logging<S, F>(make_solver: F, test_num:
usize)
where
    S: Solver,
    S::Event: Debug,
    F: FnOnce(Cnf, LoggerHandler<S::Event>) -> S
{
    <<vst_sample-cnf>>
    let cnf = sample_cnfs[test_num].clone();
    println!("problem: {:?}", test_num);
    let handler = LoggerHandler::<S::Event>::new();
    let (_result, elapsed) = run_solver(cnf, handler,
make_solver);
    println!("time: {:?}", elapsed);
}
```

```
//| id: vst_solve-many-small
```

```
//| id: vst_solve-many-large
```

```
//| file: rust/viska-sat/tests/common.rs
use viska_sat::{clause::Clause, cnf::Cnf,
event_handler::EventHandler, lit::Lit, solver::{SatResult,
Solver}};
use std::fmt::Debug;
use std::marker::PhantomData;
use std::time::{Duration, Instant};
<<vst_run-solver>>
```

```
<<vst_logger-handler>>
<<vst_solve-with-logging>>
<<vst_solve-many-small>>
<<vst_solve-many-large>>
```

4.6.2 `BruteForceSolver`

```
//| file: rust/viska-sat/tests/brute_force_solver.rs
mod common;
use common::solve_with_logging;
use viska_sat::brute_force::BruteForceSolver;

#[test]
fn brute_force_solver_with_logging() {
    for i in 0..=1 {
        solve_with_logging(|cnf, handler|
BruteForceSolver{ cnf, handler }, i);
    }
}
```

4.6.3 `DpllSolver`

```
//| file: rust/viska-sat/tests/dpll_solver.rs
mod common;
use common::solve_with_logging;
use viska_sat::dpll::DpllSolver;

#[test]
fn dpll_with_logging() {
    for i in 0..=1 {
        solve_with_logging(|cnf, handler| DpllSolver{ cnf,
```

```
handler }, i);
    }
}
```