AIPL の言語仕様の検討

— ABCL/1 の機構の再導入と、分散 AI エージェント／実時間 OS という目的から —

ABCL^c+ / AIOS 処理系（`yaskodama/abclcp`）

2026 年 07 月 30 日 09:48 JST 版

概要

先の比較レポートで、AIPL は ABCL/1 から三種のメッセージ送信を継承しつつ express mode、返信先の一級性、`where` 制約つきパターン受信を落としていることを整理した。本稿はそのうち **express mode** と `where` の二つを AIPL に入れた場合に何が起きるかを、構文・実装・型システム・形式的保証の四層で検討する。

結論を三点にまとめる。第一に、**両者は型システムをほとんど壊さない**。どちらも「いつメッセージを受理するか」の話であり、「どんな型を持つか」の話ではないためである。第二に、`where` は**進行定理に第四の枝「ガード待ち」を追加する**。これは失うものが明確で、しかも `select` と名前 dispatch の並立という既存の不整合を解消できるので、費用対効果が高い。第三に、express mode は意外にも**型健全性の証明を壊さない**。形式モデル AIPL$^-$ はアクターの逐次性を最初から落としており、同一アクターへの複数タスクの並行実行を既に許しているからである。壊れるのは**プログラム側の**不変条件であり、だからこそ ABCL/1 は atomicity 宣言を用意していた。

一方で express mode には実装上の前提がある。現在のメソッドのローカル変数はアクターの環境に直接置かれ、呼び出し時に退避・復元されていない。これは今は観測できない潜在的な問題だが、入れ子実行を導入した途端に表に出る。本稿はこれを実測で確認した。

第 II 部では視点を変え、AIPL が**分散 AI エージェント言語**と**実時間 OS（フィジカル AI）の記述言語**という二つの目的を持つとしたら、どのような言語仕様であるべきかを考察する。結論は**一つの中核と二つのプロファイル**である。二つの目的の要求は多くの点で正反対であり、一つの言語で両方を無差別に許すと、どちらの保証も得られない。両者を静的に分ける鍵は effect/capability 系であり、幸いなことに実装にはすでに 17 種の capability 分類が存在する（ただし現在は実行時メタデータにとどまり、型検査器は使っていない）。Ada の Ravenscar プロファイルという直接の先例もある。

目次

第 I 部

express mode と where の再導入

1 はじめに

1.1 問い

AIPL（ABCL^c+）は ABCL/1 の子孫を自認しているが、ABCL/1 の中核機構のうち次の三つを持っていない。

- **express mode** —- 処理中のメッセージを中断して割り込む機構。
- **返信先の一級性** —- 返信先を値として他者へ渡し、委譲する機構。
- **where** —- オブジェクトの状態に依存した受理条件。

本稿は前者と後者、すなわち**受理と割り込みに関する二つ**を導入した場合の帰結を検討する。返信先の一級化は線形型を要する別問題なので扱わない。

1.2 なぜこの二つなのか

現在の AIPL には次の具体的な穴がある。

- now で返信を待っているアクターは、その間**他のメッセージを一切処理できない**。1 アクター 1 スレッドで、await_future が条件変数でブロックするためである。状態を問い合わせることも、止めることもできない。
- 状態に依存した待ち合わせが書けない。「バッファが空でないときだけ get を受理する」を表現するには、受理してから条件を見て、駄目なら自分に送り直すしかない。受信規律の外へ出てしまう。
- 受信経路が二本ある。名前による dispatch と select が並立し、同名のメッセージが通常の dispatch に先取りされると select には残らない。

express mode は一つ目、`where` は二つ目と三つ目に直接効く。

2　提案する形

2.1　`where` ── 状態依存の受理条件

メソッド宣言と `select` の case に、真偽値の**ガード**を付けられるようにする。

```
class Buffer {
  var n = 0;
  var cap = 3;

  // n < cap のときだけ put を受理する
  method put(x) : unit where n < cap {
    n = n + 1;
    print("put, n=" ++ n);
  }

  // n > 0 のときだけ get を受理する
  method get() : int where n > 0 {
    n = n - 1;
    reply(n);
  }
}
```

`select` 側も同じ形にする。

```
select {
  case get() where n > 0 -> { reply(n); }
  timeout 500 -> { print("empty"); }
}
```

意味は ABCL/1 と同じである。ガードが偽のメッセージは**受理されず、メールボックスに残る**。状態が変わって真になれば受理される。

2.2　express mode ── 割り込み

送信側は三種それぞれに express 版を用意する。既存の `send!`（`UnsafeSend`）と衝突しないよう、修飾子として `express` を置く。

```
express send a.stop();          // 割り込んで実行、待たない
var s = express now a.status(); // 割り込んで実行、返信を待つ
var f = express future a.status();
```

受信側は、割り込みで実行されてよいメソッドを明示する。どのメソッドが他のメソッドの実行を中断しうるかは、読み手にとって最も重要な情報だからである。

```
class Worker {
  var progress = 0;
  var stopped  = 0;
```

```
  // 通常のメソッド。長く走る
  method run(n) : unit {
    var i = 0;
    while i < n do {
      if (stopped >= 1) { i = n; }
      progress = progress + 1;
      i = i + 1;
    }
  }

  // 割り込みで呼ばれてよい。run の実行中でも応答する
  express method status() : int {
    reply(progress);
  }

  express method stop() : unit {
    stopped = 1;
  }
}
```

割り込まれたくない区間は `atomic` で保護する。

```
method transfer(k) : unit {
  atomic {
    from = from - k;      // この2行の間に割り込まれると
    to   = to + k;        // 合計が壊れる
  }
}
```

3　実装への影響

AIPL の実装を読んだうえで、変更箇所を具体的に見積もる。

3.1　`where` の実装

ファイル	変更
`lexer.mll`	`where` をキーワードに追加
`parser.mly`	`method_decl` に省略可能な `where expr`、`select_case` にも同じものを追加
`ast.ml`	`method_decl` に `guard : expr option`、`select_case` にも同様
`infer.ml`	ガードをメソッドの環境（フィールド＋仮引数）で `bool` として型検査するだけ
`eval_thread.ml`	**ここが本体**。`actor_loop` が `Queue.pop` で先頭を取るのをやめ、メールボックスを走査して「名前が一致し、かつガードが真」の最初のメッセージを選ぶ

走査の機構はすでにある。`select` の実装 `pop_matching_message` はメールボックスを一旦すべて取り出し、パターンに合うものを探し、残りを戻している。ガード評価を足すだけで再利用できる。

注意 1. これは select と通常 dispatch を**統一する好機**でもある。通常 dispatch がガード付きの走査になるなら、select は「この名前の集合のみを対象に、タイムアウト付きで走査する」という特別な場合にすぎない。受信経路が一本になり、先取り競合という現在の不整合が消える。

3.2 express mode の実装

ファイル	変更
lexer.mll	express, atomic をキーワードに追加
parser.mly	送信の express 版 3 種、express method、atomic { ... } 文
ast.ml	送信に mode を持たせる、method_decl に express : bool、Atomic of stmt
types.ml / infer.ml	express メソッドの reply を、選択されたメッセージの ρ に束縛する（select の case で既に行っている処理と同型）
eval_thread.ml	アクターに第二のキューを足す。eval_stmt の文境界で express キューを覗き、あれば**入れ子で**評価する。atomic 区間ではこの覗き込みを止める

割り込みは協調的なポーリングで実現する。OCaml のシステムスレッドを外から割り込むのは現実的でないので、eval_stmt が文の境界で express キューを確認し、あればその場で express メソッドの本体を入れ子に評価して戻る。粒度は文単位になる。

3.3 前提となる修正 —- ローカル変数のスコープ

入れ子実行を入れる前に直さなければならない箇所がある。

測定 1. 現在、メソッドの**仮引数**は呼び出しの前後で退避・復元されている。

```
let saved = List.map (fun p -> (p, Hashtbl.find_opt actor.env p)) params in
  List.iter2 (fun p v -> Hashtbl.replace actor.env p v) params arg_vals;
  ...
  eval_stmt actor mdecl.body;
  List.iter (fun (p, ov) -> ...) saved      (* 復元 *)
```

しかし本体の var で宣言された**ローカル変数**は eval_stmt が actor.env へ直接書き込むだけで、退避も復元もされない。

これは**今は観測できない**。型検査器が各メソッドを独立した環境（clone env）で検査するため、他のメソッドのローカルを参照するプログラムは型エラーになる。実測すると次のとおりである。

```
class C {
  method a() : unit { var tmp = 777; print("a: tmp=" ++ tmp); }
  method b() : unit { print("b: tmp=" ++ tmp); }   // 宣言していない
}
```

```
TYPE_ERROR: line 8, col 24: unbound variable: tmp
```

型が守っているので実害が出ていない。しかし express mode を入れると**同一アクター上で入れ子実行が起きる**。外側のメソッドが var tmp を持ち、割り込んだ express メソッドも var tmp を持てば、両者は型検査上まったく別の変数なのに実行時には同じ actor.env のエントリを共有し、上書きされる。

したがって express mode の前提として、**メソッド呼び出しごとのフレームを導入する**（またはローカル変数も仮引数と同様に退避・復元する）修正が必要である。これは express mode とは独立に価値のある修正でもある。

4　型システムへの影響 —- ほぼ無傷

ここが本稿の中心的な観察である。

4.1　どちらも「型」の話ではない

`where` も express mode も、**メッセージをいつ受理するか**を変える機構であって、**メッセージがどんな型を持つか**を変えない。AIPL の型システムが述べているのは後者だけである。

現在の検査	二機構の影響
メソッドの存在	影響なし
引数の個数と型	影響なし
返信の型	影響なし
返信の回数（高々一度）	express は**文脈の分離**が必要。`select` の case で既に実装済みの処理と同型
`select` の照合	`where` でガードの `bool` 検査が増えるだけ
公開時の注釈	影響なし

4.2　`where` の型付け

ガードはメソッドの環境（フィールド＋仮引数）で `bool` として型検査するだけである。追加規則は次の一行で足りる。

$$\frac{\Gamma, \overline{x{:}\tau} \vdash g : \texttt{bool} \qquad \Gamma, \overline{x{:}\tau} \vdash_{C.m} body}{\Gamma \vdash \texttt{method}\ m(\overline{x})\ \texttt{where}\ g\ \{body\}}\ \textsc{Method-Guard}$$

依存型や篩型は**不要**である。それらが必要になるのは「ガードが真なら本体は安全」を**証明**したい場合であって、型安全性のためではない。

4.3　express mode における `reply` の帰属

express メソッドの本体にある `reply` は、割り込んだ express メッセージへの返信である。囲んでいた通常メソッドの返信ではない。

これは既に解決済みの問題と同型である。`select` の case 本体の `reply` も、囲むメソッドではなく選択されたメッセージへの返信であり、型検査器は case 本体で `reply` をそのメソッドの ρ に束縛し直している。express メソッドも同じ扱いでよい。実行時に `ctx_msg_id` を退避・復元する処理も、`select` の実装がそのまま雛形になる。

4.4 ガードの純粋性という新しい要求

一点だけ、型システムに新しい要求が生じる。**ガードは副作用を持ってはならない**。受理の可否を試すためにガードを評価するので、そこで状態が変わると「試したら受理条件が変わった」ことになる。

現在の OCaml 実装には効果システムが無い。最小限の対処は構文的な制限である。

- ガードで参照できるのはフィールドと仮引数のみ。
- 呼び出せるのは純粋なプリミティブのみ（比較、算術など）。
- `now` / `future` / `send` / `reply` / `new` / `become` は書けない。

これは既存の構文検査（`expr_has_reply` など）と同じ形の走査で実装できる。

5 形式的保証への影響

5.1 保存定理 —- 影響なし

保存（preservation）は「well-typed な構成が一歩進んでも well-typed」である。状態の型不変条件は「状態が常に宣言された型を持つ」ことしか言わない。`where` は遷移が起こる条件を狭めるだけ、express mode は遷移の順序を変えるだけなので、どちらも保存を壊さない。

5.2 進行定理 —- `where` は枝を一本増やす

現在の進行定理は三択として述べられている —- 値・簡約・`await` 待ち。デッドロックは型では排除できないので、その境界を明示したものである。

`where` を入れると、**第四の枝が必要になる**。メールボックスにメッセージがあるのに、どのガードも真でないため配送できない構成が存在するからである。

現在	`where` 導入後
値である	値である
簡約できる	簡約できる
`await` 待ちである	`await` 待ちである
	ガード待ちである

これは失うものが明確である。定理の主張が一つ弱くなる代わりに、「なぜ止まっているか」の分類が一つ増える。言い換えれば、`where` は**デッドロックの形を一つ増やす**。バッファが空のまま `get` だけが溜まれば、それは正当な待ちであるが、プログラムの誤りでそうなった場合と静的に区別できない。

5.3 逐次性 —- express mode は証明を壊さない

これは直観に反するが、確認できた事実である。形式モデル AIPL^{-} は「保証しないこと」の一覧で、アクターの逐次性を明示的に**落として**いる。

> actor の逐次性 —- 落とした。同じ actor あての二通が同時にタスクになりうる。

配送規則は無条件に新しいタスクを起こすので、同一アクターに対して複数のタスクが並行に走りう

る。そして同レポートは、これが**型安全性には影響しない**と明言している。状態の型不変条件は状態が常に正しい型であることしか言わず、どの順で書き換えられるかは問わないからである。

したがって express mode の割り込みは、**形式モデルが既に許している実行の一部**にすぎない。逆に言えば、現在の**実装**が形式モデルより厳しく（1 アクター 1 スレッドの逐次実行）作られているだけである。express mode はその差を埋める方向の変更であり、型健全性の証明をやり直す必要はない。

5.4 危ういのはプログラム側の不変条件

壊れるのは型ではなく、プログラムが持っている**意味的な**不変条件である。

リポジトリの健全性レポートは哲学者の食事問題について資源デッドロック自由を証明しているが、その証明はフォーク割り当てプロトコルの抽象状態機械 $s = (\mathrm{phase}, \mathrm{own})$ についてのものであり、状態の健全性 $\mathrm{wf}(s)$ は「Hold なら `lo` のみを持つ」のように **phase と own が整合していること**を要求する。

express mode で「phase を書き換えたが own はまだ」という瞬間に割り込まれると、その瞬間の状態は $\mathrm{wf}(s)$ を満たさない。型は壊れないが、抽象状態機械との対応が壊れる。同レポートが「実装との対応は言えない」と述べている箇所が、まさにここで効いてくる。

これが ABCL/1 に atomicity 宣言があった理由である。割り込みを許すなら、割り込まれてはいけない区間を宣言できなければならない。express mode と `atomic` は分離できない一対の機構である。

6 メリット

6.1 `where` の利点

1. **同期が宣言的になる。**有界バッファ、資源プール、段階制御（初期化が済むまで受け付けない）などが、受理条件としてそのまま書ける。フラグ変数も待ちループも要らない。
2. **「受理してから条件を見て送り直す」という悪い定石が消える。**現在この形を書くと、条件が偽の間メールボックスを自分で埋め続けることになる。受信規律の中で待てるようになる。
3. **受信経路が一本化できる。**通常 dispatch がガード付き走査になれば、`select` はその特別な場合になる。現在の「`select` が通常 dispatch に先取りされる」という観測可能な不整合が消える。これは表現力を増やす話ではなく、後付けで生じた矛盾の解消である。
4. **既存の機構を再利用できる。**メールボックスの走査は `select` の実装にすでにある。ガード評価を足すだけである。
5. **型システムをほとんど触らない。**ガードの `bool` 検査が増えるだけで、依存型は要らない。

6.2 express mode の利点

1. **塞がったアクターに手が届く。**現在 `now` で待っているアクターは石になる。express mode があれば、待っている最中でも状態問い合わせ・中止指示・監視に応答できる。これは現在の AIPL の最大のモデル的な穴に直接効く。
2. **デッドロックからの脱出手段になる。**相互 `now` で固まった二者に、外から `express send a.abort()` を送れる。型では排除できないデッドロックに対する**運用上の**答えになる。
3. **監視・内省が現実的になる。**`aios_*` の内省プリミティブは、対象アクターが動いていると意味のある値を返しにくい。express mode があれば「いま何をしているか」を当人に聞ける。

4. **タイムアウトの段階的な扱いが書ける。**`select` の `timeout` は自分が待つ側の機構だが、express mode は**相手を叩く**側の機構である。両方あると「一定時間返らなければ問い合わせ、それでも駄目なら中止」が書ける。
5. **型健全性の証明を壊さない**（§5.3）。

7 デメリットと危険

7.1 `where` の代償

項目	内容
進行定理が弱まる	「ガード待ち」という第四の枝が増える（§5.2）
FIFO でなくなる	メッセージの処理順が到着順と一致しなくなる。意味論として明記が必要
飢餓	あるメッセージのガードが永遠に真にならない場合、そのメッセージは永久に残る。公平性は保証できない
静的に検出できない誤り	恒偽のガードを持つメソッドは「決して受理されない」が、型検査は通る。`select` の arity 不一致と同じ種類の**静かに壊れる**誤りが一つ増える
コスト	配送ごとにメールボックスを走査し、各メッセージについてガードを評価する。$O(1)$ から $O(n)$ になる
ガードの純粋性	効果システムが無いので構文的に制限する必要がある（§4.4）

飢餓と恒偽ガードについては、緩和策がある。明らかに恒偽なガード（`false` や定数比較）は警告できる。また、あるメッセージが一定回数の走査で受理されなかったことを実行時に記録すれば、診断の手がかりになる。

7.2 express mode の代償

項目	内容
意味的不変条件が壊れる	最大の問題。`atomic` と対で導入しなければ使い物にならない（§5.4）
前提となる修正	メソッド呼び出しごとのフレームが必要。現在ローカル変数は `actor.env` に直接置かれ、退避・復元されていない（§3.3、実測済み）
割り込み粒度	協調的ポーリングなので文境界でしか割り込めない。単一の長い式や組込み関数の内部では割り込めない
再入	express メソッドが通常メソッドを呼ぶ、express の中で express が来る、といった入れ子の深さ制限が要る
デバッグ可能性	実行の非決定性が増える。「どこで割り込まれたか」を再現するのが難しい
既存プログラムの前提	「メソッドは中断されない」を暗黙に仮定している既存コードは、express メソッドを一つ足した時点でその仮定を失う。既定は非割り込みにすべき
証明との対応	型健全性は無傷だが、抽象状態機械との refinement を主張していた箇所は `atomic` の裏付けが必要になる

7.3 二機構の相互作用

- **express メッセージにガードを許すか。**許すと「割り込みたいのに受理されない」という状況が生じ、express の目的（緊急の応答）に反する。**express メソッドにはガードを付けられない**とするのが妥当である。
- **`atomic` 区間の中で `now` を書けるか。**書けると、`atomic` のまま返信を待つことになり、その間割り込めない。express mode の目的を損なう。**`atomic` 区間では `now` と `await` を禁止**するのが安全である。これは構文的に検査できる。
- **ガード評価中に割り込むか。**ガードは純粋で短いので、割り込まないのが単純である。

8 段階的な導入案

いずれも独立に価値があり、この順序なら各段階で退行を測れる。

段階	内容	備考
0	メソッド呼び出しフレームの導入	ローカル変数を `actor.env` から出す。express mode の前提だが単独でも意味がある潜在バグの修正
1	`select` の case に `where`	最小。通常 dispatch の FIFO を変えないので影響が閉じている。ガードの `bool` 検査と純粋性検査をここで作る
2	メソッドに `where`、通常 dispatch を走査化	進行定理に枝が増える。`select` を統一する好機
3	`atomic` 区間	express より**先**に入れる。中で `now` / `await` を禁止する検査も同時に
4	express mode	段階 0 と 3 が前提。既定は非割り込み、`express method` と明示したものだけ割り込める

段階 1 と 2 の間、段階 3 と 4 の間で、既存コーパスの型検査判定と差分テストハーネスを回して測るべきである。これまでの改修で、見積りが実測と食い違うことが繰り返しあった。

9 結論

9.1 どちらを先に入れるべきか

`where` を先に入れるべきである。理由は三つある。

1. 型システムへの影響が `bool` 検査の追加だけで済む。
2. 失うものが「進行定理に枝が一本増える」と明確に述べられる。
3. `select` と名前 dispatch の並立という**既存の**不整合を解消できる。表現力の追加ではなく、負債の返済としての価値がある。

express mode は、前提となる修正（呼び出しフレーム）と対で必要な機構（`atomic`）を抱えており、しかも既存コードの暗黙の仮定を壊す。価値は高いが、`where` のあとに置くのが順序として自然である。

9.2　全体としてどう評価するか

この二つを入れると、AIPL は ABCL/1 に確かに近づく。そして**その代償は型システムではなく、推論の難しさとして現れる**。

型健全性は無傷である。保存は影響を受けず、逐次性は形式モデルが最初から落としているので express mode も証明を壊さない。進行定理だけが `where` によって枝を一本増やす。

しかし、プログラムについて**個別に**証明していた性質—- 哲学者の食事問題の資源デッドロック自由のような —- は、割り込みが入ると抽象状態機械との対応を `atomic` で裏づけ直す必要がある。

先の比較レポートで「失った表現力が型付けを容易にしている」と書いた。本稿はその裏返しである。**表現力を戻すと、型は保てるが、個別の証明の手間が増える**。どちらを重く見るかが判断の分かれ目であり、研究目的が「実装された言語の健全性を機械検証する」ことにあるなら、`where` までは進み、express mode は `atomic` の設計が固まってから、というのが妥当な線である。

第 II 部
二つの目的に向けた言語仕様の考察

10　二つの目的とその要求

ここから視点を変える。AIPL が次の二つを目的に持つとしたら、どのような言語仕様であるべきか。

- **分散 AI エージェント言語** —- 複数ノードに分散したエージェントが、モデル呼び出し・記憶・タスク・プロトコルを通じて協調する。実装にはすでに `ai_call`、`model_generate`、`aios_memory_*`、`aios_task_*`、`protocol_*`、`remote(...)`、HTTP ゲートウェイが揃っている。
- **実時間 OS（フィジカル AI）の記述言語** —- センサ・アクチュエータを持つ実機の上で、周期処理と割り込みを記述する。別リポジトリで Xinu 上の実行系（プリエンプティブ優先度、tickless タイマ）と AIPL からのコード生成が進んでいることを前提とする。

10.1　分散 AI エージェント言語としての要求

1. **待ち時間が長く、しかも予測できない。**モデル呼び出しは秒単位で、失敗もする。`now` で待つと、その間アクターは何もできない。
2. **失敗が普通である。**タイムアウト、部分的な成功、リトライが日常であり、「返るか返らないか」ではなく「何が返り、返らなかったときどうするか」を書けなければならない。
3. **結果が構造を持つ。**モデルの出力は JSON 相当の構造である。現在の AIPL の型は `int` / `float` / `string` / `bool` / `unit` / 配列 / アクターだけで、**レコードも直和も無い**。構造をすべて文字列で運ぶことになる。
4. **どこへ何が出ていくかが重要である。**機外へ出る呼び出しとローカル推論は区別されなければならない。監査・秘匿・課金のいずれの観点からも必要である。
5. **対話の順序に規律がある。**`protocol_*` はすでにセッションプロトコルを実行時に検査しているが、型ではない。

10.2 実時間 OS（フィジカル AI）としての要求

1. **時間が仕様である。**周期、期限、最悪実行時間、ジッタが正しさの一部になる。「いつか終わる」では不十分である。
2. **有界であること。**メモリ、ループ回数、メールボックス長、再帰深さ —- すべてに上限が要る。動的割り付けは初期化後には禁止したい。
3. **割り込みが本質である。**センサ・タイマ・通信の到着は割り込みであり、第 I 部の express mode がまさにこれである。そして割り込まれてはいけない区間（`atomic`）が必ず要る。
4. **ブロックが致命的である。**`now` で無期限に待つ制御ループは期限を守れない。
5. **数値の決定性。**浮動小数はハードウェアによって時間も結果も揺れる。制御ループでは整数か固定小数点が望ましい。なお AIPL では `/` が `int` 同士でも `float` を返す仕様である。
6. **物理量の取り違えが事故になる。**`m/s` と `rad/s` を足しても現在の型システムは何も言わない。

10.3 両者は多くの点で正反対である

	分散 AI エージェント	実時間 OS
待ち時間	秒単位、予測不能	マイクロ秒〜ミリ秒、上界必須
失敗	日常。設計に織り込む	許されない（設計で排除する）
データ	構造化・可変長・スキーマ推測	固定長・既知
メモリ	動的でよい	初期化後は静的
非決定性	本質的（モデル出力）	排除したい
`now`	使いたいが長く待つ	期限なしでは使えない
浮動小数	問題ない	避けたい
外部通信	中心的な機能	限定・帯域保証つき
優先度	概念が薄い	中心的

同じ言語で両方を**無差別に**許すと、どちらの保証も得られない。実時間側は「この部分は割り付けもブロックもしない」と言えなくなり、エージェント側は不要な制約を負う。

11 提案 —- 一つの中核と二つのプロファイル

11.1 先例 —- Ada の Ravenscar プロファイル

この形には直接の先例がある。Ada は汎用言語だが、高信頼・実時間用途に対して **Ravenscar プロファイル**という制限された部分集合を定義した。動的なタスク生成を禁じ、待ち合わせの形を限定し、解析可能性を確保する。言語を分けるのではなく、**同じ言語の中に検証可能な部分集合を切る**という設計である。

同様に、分散側には E 言語という先例がある。E は**ブロックを一切許さず**、eventual send と promise だけで書かせる。デッドロックを言語から排除するための選択である。

AIPL が採るべきはこの二つを合わせた形である。

	プロファイル A（agent）	プロファイル R（realtime）
方針	ブロックを避け、失敗を型に持つ	有界性と期限を静的に保証する
許す effect	net, ai, fs, mem, time	io（宣言した装置のみ）, time
禁止	無期限 now	now（期限なし）、動的割り付け、非有界ループ、再帰、浮動小数
必須	失敗の扱い、リモートインタフェース宣言	周期／期限／優先度の宣言、atomic

11.2 すでにある土台 —- capability 分類

好都合なことに、実装にはすでに capability の分類が入っている。プリミティブごとに分類が付けられ、実行時に列挙できる。実測すると 17 分類あった。

```
Core.Math  Core.Array  Core.Introspection  Console  Time
Actor  Actor.Introspection
AIOS.Kernel  AIOS.Service  AIOS.Task  AIOS.Memory  AIOS.Event
AIOS.Model  AIOS.Model.OpenAI  AIOS.Model.Gemini  AIOS.Remote
Protocol.Session  UI.SDL  Web
```

測定 2. この分類は**実行時メタデータにとどまっており、型検査器は一切使っていない**。`capabilities()` と `capability_prims(c)` で内省できるだけである。

つまりプロファイル分離のための**データはすでにある**。足りないのは、これを静的な effect として**メソッドに伝播させ、宣言と照合する**機構だけである。姉妹実装（Python 版 AIPL）では効果注釈が gradual に検査されており、方式の前例も内部にある。

11.3 effect を静的にする

メソッドに effect を宣言でき、本体が使うプリミティブと送信先の effect の和が宣言に含まれることを検査する。

```
class Planner {
  // 機外へ出ることを宣言している
  method plan(goal) : string !{ai, net} {
    reply(ai_call("plan: " ++ goal));
  }
}

class Controller {
  // io と time だけ。net も ai も使えない
  method step(sensor) : int !{io, time} {
    reply(clamp(sensor));
  }
}
```

これで得られるものは大きい。

- **機外に出ないローカル推論を型で示せる。**たとえばオンデバイス推論のプリミティブに `{ai}` だけを与え、`net` を含めなければ、「AI を使うが機外へは出ない」ことがシグネチャに現れる。

- **実時間側で禁止すべき effect を列挙できる。**`net` や `ai` を持つメソッドはプロファイル R では呼べない、と機械的に言える。
- **監査ができる。**どのアクターがどの外部資源に触るかが静的に集計できる。

12　中核に必要な仕様

両プロファイルに共通して必要なものを挙げる。いずれも現在の穴を埋める作業でもある。

1. **型の穴を閉じる。**`sender` が `any` で送信が無検査、リモート送信先が `any`、アクター型が単一化で区別されない、`become` が置換後のインタフェースを検査しない —- これらは分散でも実時間でも致命的である。分散では相手を間違え、実時間では存在しないメソッドへ送っても実行時まで分からない。
2. **リモートインタフェース記述。**対向ノードの `class C { method m(x) : T }` だけを書いた記述を読み込み、`remote(...)` に型を付ける。戻り値型注釈を必須にしたのはこの前提づくりであった。実時間メッシュでもエージェント連携でも、これが無いと境界が `any` のままである。
3. **レコードと直和。**現在は構造化データを表現できない。エージェント側は JSON 相当、実時間側はセンサフレームやコマンドを構造として運びたい。直和があれば「成功か失敗か」も型で表せる。
4. **`where` ガード**（第 I 部）。エージェント側では「必要なデータが揃うまで受理しない」、実時間側では「状態機械の段階に合わない指令を受理しない」に効く。どちらも本質的である。
5. **`==` と `!=`。**現在これらは字句解析器と評価器にあるのに**文法規則が無く書けない**。状態機械を書く言語としては欠落が大きい。

13　プロファイル A —- 分散 AI エージェント

13.1　無期限 `now` を禁止し、期限付きにする

エージェントにとって最悪の事態は、モデル呼び出しの返りを待ってアクターが石になることである。`now` を廃するのではなく、**期限と失敗時の分岐を必須にする**。

```
var r = now planner.plan(goal) timeout 5000 else "fallback";
```

型としては `now ... timeout ... else e` の全体が ρ 型の式であり、`else` 節も同じ型を持つことを要求すればよい。既存の `select` の `timeout` と同じ機構で実装できる。

これは進行定理に対して**良い方向**に働く。現在の進行定理の第三の枝「`await` 待ち」は、期限付き `now` だけを使うプログラムでは有限時間で解消するので、その範囲でデッドロック自由が言える。言い換えれば、**期限を必須にすることは型で保証できる進行性を増やす**。

13.2　失敗を型に持つ

期限切れ、モデルの拒否、ネットワーク断は日常である。直和があれば戻り値型として書ける。

```
method plan(goal) : Result<string, Error> !{ai, net} { ... }
```

現在は「失敗したら文字列に "ERROR: ..." を入れる」しかなく、呼び出し側が確認を忘れても型検査は通る。

13.3 セッションプロトコルを型へ上げる

protocol_* は実行時にプロトコル状態を検査している。これを型レベルのセッション型へ上げることは、健全性レポートがすでに次の課題として挙げている。戻り値型注釈を必須にしたことで、$!m(T).?R.\cdots$ の R がインタフェースに書かれている状態になった。前提は整っている。

13.4 構造化出力のスキーマ検査

モデル出力を型付きの値へ取り込む箇所は、必ず動的検査になる。そこを**一点に閉じ込める**のが型システムの役目である。

```
// 検査に失敗したら Err になる。以降は型が保証される
var p = parse<Plan>(now planner.plan(goal) timeout 5000 else "");
```

14 プロファイル R —- 実時間 OS ／フィジカル AI

14.1 時間を宣言する

周期、期限、優先度をメソッドの宣言に置く。

```
class Balancer {
  // 1kHz、期限 800us、優先度 10
  periodic method tick() : unit !{io, time}
      period 1000us deadline 800us priority 10 {
    var s = read_gyro();
    set_motor(pid(s));
  }
}
```

これは型システムの仕事ではなく**スケジューリング可能性解析**の仕事だが、言語が宣言を持てば解析器を書ける。Xinu 上の実行系がプリエンプティブ優先度と tickless タイマを持つなら、宣言をそのままスケジューラの設定へ落とせる。

14.2 有界性を静的に強制する

プロファイル R では次を禁止または制限する。

対象	扱い
new（動的アクター生成）	初期化フェーズのみ許可
配列の伸長（array_push）	禁止。容量は宣言する
while	禁止。上界の明らかな for だけを許す
再帰・自分あて送信の連鎖	禁止（呼び出しグラフで検査）
期限なし now / await	禁止
浮動小数	禁止または固定小数点に置換
net, ai, fs effect	禁止

これらはいずれも**構文と effect の検査で機械的に実施できる**。依存型は要らない。`while` の禁止は既存の `reply` 線形性検査（ループ本体の `reply` を「2 回以上」と判定する走査）と同じ形の解析で足りる。

14.3 数値 —- 固定小数点と単位

浮動小数を避けるなら、`int` を基本にした固定小数点型が要る。ここで、本セッションで整理した数値の扱いが直接効いてくる。現在 `/` は `int` 同士でも `float` を返す。プロファイル R ではこれを許せないので、`div` と `mod`、あるいは固定小数点の `/` を別に用意することになる。

さらにフィジカル AI では**物理単位**を型に載せる価値が大きい。

```
var v : float<m/s>   = 1.2;
var w : float<rad/s> = 0.3;
var x = v + w;          // 単位不一致として型エラーにしたい
```

単位は型の添字として扱えるので、依存型は不要である。乗除で指数を足し引きする規則を単一化に入れるだけで済み、制御ソフトの事故の実際的な原因を一つ潰せる。

14.4 割り込みとしての express mode

第 I 部の express mode は、実時間側では選択肢ではなく**必然**である。センサ・タイマ・通信の到着は割り込みであり、`express method` がハンドラ、`atomic` がクリティカルセクションに対応する。

そして第 I 部で挙げた前提と制約が、そのまま実時間の作法になる。

- メソッド呼び出しフレームの導入（割り込みハンドラは入れ子で走る）。
- `atomic` 区間では `now` / `await` を禁止。
- ハンドラ自身も有界でなければならない。**express メソッドはプロファイル R の制約を必ず満たす**—- ループ禁止、割り付け禁止、期限つき —- とすべきである。

14.5 有界メールボックスと溢れ方針

無限に伸びるキューは実時間では使えない。容量と、溢れたときの方針を宣言させる。

```
class Sensor mailbox 8 overflow drop_oldest { ... }
```

溢れが起きうることは型では消せないが、**どうするかを宣言させる**ことで、暗黙の失敗を無くせる。

15 二つのプロファイルの境界

分散エージェントと実時間制御は同じシステムの中で協調する—- エージェントが判断し、実時間側が実行する。両者をつなぐ規律が要る。

15.1 方向の規律

方向	許す送信	理由
A → R	`send` のみ	実時間側の応答を待つとエージェント側が塞がる。結果が要るなら R 側から通知させる
R → A	期限付き `future` のみ	実時間側は決してブロックしてはならない。`now` は使えない

すなわち**実時間側からエージェント側への同期呼び出しを型で禁止する**。これは effect と期限の検査で機械的に実施できる。「制御ループが LLM の返りを待って期限を落とす」という最も起こりやすい事故を、言語が構造的に防ぐ。

15.2 境界アクター

境界はアクターとして明示する。

```
// エージェント側の判断を受け取り、実時間側へ渡すだけの境界
class Bridge profile boundary {
  var target = 0;

  // A 側からは send のみ
  express method set_target(v) : unit !{} { target = v; }

  // R 側は即座に読める
  method get_target() : int !{} { reply(target); }
}
```

境界アクターは**状態の受け渡しだけを行い、計算しない**。express で書き込み、通常メソッドで読む。これなら実時間側は待たない。

16 型健全性と証明への影響

提案した仕様が、既存の形式的保証にどう影響するかを整理する。

仕様	影響
effect 系	保存・進行に影響なし。effect は型の**添え物**として独立に定式化できる（型と effect の対で保存を述べる標準的な形）
レコード・直和	標準的な拡張。保存の証明が場合分けで長くなるだけ
期限付き `now`	**進行定理を強める**。期限付き待ちは有限時間で解消するので、第三の枝が「有限時間後に解消する待ち」に格上げできる
`where`	進行定理に「ガード待ち」の枝を追加（第 I 部）
express + `atomic`	型健全性は無傷。プログラム個別の不変条件は `atomic` で裏づけ直す必要がある
有界性の制約	型システムの外側の構文・グラフ解析。健全性の証明には影響しないが、**停止性**という現在言えていない性質をプロファイル R では言えるようになる
物理単位	型の添字として単一化に入れるだけ。依存型は不要

注目すべきは、**プロファイル R では現在言えていない性質が言えるようになる**ことである。健全性レポートは「停止性は言えない —- メソッドが自分あてに送れば無限に走る」と述べているが、プロファイル R はまさにそれを禁止するので、停止性と応答時間の上界が主張できる。

17　実装の順序

これまでの改修で、見積りが実測と食い違うことが繰り返しあった。以下は順序の提案であり、各段階で既存コーパスの判定と差分テストハーネスを回して測るべきである。

	内容	位置づけ
1	`==` / `!=` の文法追加	数行。欠落の解消
2	型の穴を閉じる（`sender`、アクター型の区別、`become`）	両プロファイルの前提
3	effect 注釈と検査（既存 capability 分類を静的化）	**プロファイル分離の土台**。ここが要
4	リモートインタフェース記述	分散の境界を閉じる
5	レコードと直和	エージェント側の構造化データ、失敗の型
6	期限付き `now`	進行性が強まる。A の必須要件
7	`where` ガードと受信経路の統一	第 I 部
8	呼び出しフレーム、`atomic`、express mode	第 I 部。R の必須要件
9	有界性の制約とプロファイル宣言	R の閉じ込め
10	周期・期限・優先度の宣言とスケジューリング解析	Xinu 実行系との接続
11	物理単位	あれば事故が減る。優先度は低い
12	セッション型	長期目標

段階 3 が要である。effect が静的になれば、プロファイルの区別も、境界の方向の規律も、実時間側の禁止事項も、すべて同じ機構の上に載る。そして既存の capability 分類がそのまま使えるので、分類を新たに設計する必要はない。

18　まとめ —- どのような言語仕様とすべきか

1. **一つの中核と二つのプロファイル**にする。二つの目的の要求は正反対であり、無差別に許すとどちらの保証も得られない。Ada の Ravenscar プロファイルという先例がある。
2. **プロファイルを分ける機構は effect/capability 系**である。実装にはすでに 17 種の capability

分類があり、実行時メタデータにとどまっている。これを静的にすることが最優先である。

3. **中核では型の穴を閉じ、構造化データを入れる。**`sender` と `remote` の `any`、アクター型の非区別、`become` の未検査、レコード・直和の不在、`==` の欠落 —- これらは分散でも実時間でも支障になる。
4. **エージェント側は「ブロックしない」を型で強制する。**期限付き `now` を必須にし、失敗を直和で表す。これは進行定理を**強める**方向の変更である。
5. **実時間側は「有界である」を静的に強制する。**ループ・割り付け・再帰・浮動小数・外部通信を制限し、周期と期限を宣言する。express mode と `atomic` は選択肢ではなく必然である。そしてこのプロファイルでは、現在言えていない停止性と応答時間の上界が主張できるようになる。
6. **境界は方向を規律する。**実時間側からエージェント側への同期呼び出しを型で禁止する。「制御ループがモデルの返りを待って期限を落とす」という最も起こりやすい事故を、言語が構造的に防ぐ。

第 I 部で「表現力を戻すと、型は保てるが個別の証明の手間が増える」と書いた。第 II 部の結論はその続きである。**目的を二つに分けて宣言させれば、それぞれの目的に見合った保証を別々に主張できる。**一つの言語で一つの保証を目指すより、一つの中核で二つの保証を目指すほうが、この二つの目的には適している。

第 III 部
AIPL 理想仕様（案）

第 II 部の考察を、仕様として明文化する。以下は「こうあるべき」という案であり、現状の実装とは異なる。現状との差分は §25 にまとめる。

19　設計原則

P1 アクターが唯一の並行単位である。共有変数を持たない。状態はアクターに閉じ、外からはメッセージでしか触れない。

P2 通信は三種、そして待つなら期限を持つ。`send`（待たない）、`now`（待つ）、`future`（後で受け取る）。**無期限に待つ構文は存在しない。**

P3 返信はちょうど一度、宣言された型で行う。戻り値型は必ず宣言する。推論は補助であって省略の許可ではない。

P4 受理は名前とガードだけで決まり、受信経路は一本である。同じメッセージが二つの経路で競合しない。

P5 効果は宣言し、検査する。プロファイルは効果と構文の制限として定義される。新しい概念を足すのではなく、既にある capability 分類を静的にする。

P6 境界はインタフェース宣言でのみ型付ける。`any` は仕様に存在しない。リモートも外部公開も、宣言された interface を通す。

P7 実時間プロファイルでは有界性を静的に強制する。ループ、割り付け、再帰、待ち、外部通信のすべてに上限か禁止を課す。

20　構文

20.1　宣言

```
program    ::= decl*
decl       ::= class | interface | record | variant | global

class      ::= "class" ID profile? mailbox? "{" field* method* "}"
profile    ::= "profile" ("agent" | "realtime" | "boundary")
mailbox    ::= "mailbox" INT overflow?
overflow   ::= "overflow" ("drop_oldest" | "drop_newest" | "error")

interface  ::= "interface" ID "{" sig* "}"
sig        ::= "method" ID "(" params ")" ":" type effects? ";"

record     ::= "record" ID "{" ID ":" type ("," ID ":" type)* "}"
variant    ::= "variant" ID "{" ID ("(" type ")")?
                          ("|" ID ("(" type ")")?)* "}"

field      ::= "var" ID ":" type "=" expr ";"
method     ::= "express"? "method" ID "(" params ")" ":" type
               effects? guard? timing? block
params     ::= (ID ":" type ("," ID ":" type)*)?
effects    ::= "!" "{" effect ("," effect)* "}"
guard      ::= "where" expr
timing     ::= ("period" dur)? ("deadline" dur)? ("priority" INT)?
dur        ::= INT ("us" | "ms" | "s")
global     ::= stmt
```

現状との主な違いは、**仮引数とフィールドに型を書く**こと、**戻り値型が必須**であること、そして `interface` / `record` / `variant` / `profile` / `mailbox` / `express` / `where` / `timing` が加わることである。

20.2　型

```
type ::= "int" | "bool" | "string" | "unit"
       | "float" | "float" "<" unit ">"        // 物理単位つき
       | "fix" "<" INT "," INT ">"             // 固定小数点（整数部,小数部ビット）
       | type "[" INT "]"                      // 固定長配列
       | type "[]"                             // 可変長配列（プロファイル A のみ）
       | "future" "<" type ">"
       | ID                                    // actor / interface / record / variant
       | ID "<" type ("," type)* ">"           // 型構築子の適用
unit ::= ID ("^" INT)? (("*"|"/") ID ("^" INT)?)*   // m, m/s, rad/s, kg*m/s^2
```

`any` **は無い。**現状 `any` が使われている三箇所は次のように置き換える。

現状	理想仕様
sender : any	sender : Reply<ρ>（返信のみ可能な能力）
リモート送信先 : any	宣言された interface 型
未注釈メソッドの戻り値 : any	戻り値型は必須なので存在しない

20.3 式

```
expr ::= INT | FLOAT | STRING | "true" | "false" | ID
       | expr binop expr | "!" expr
       | expr "." ID                                  // レコードのフィールド
       | ID "{" ID "=" expr ("," ID "=" expr)* "}"    // レコード生成
       | ID "(" args ")"                              // 関数・構築子
       | "new" ID "(" args ")"                        // プロファイル A / 初期化のみ
       | "now" target "." ID "(" args ")" "timeout" dur "else" expr
       | "express" "now" target "." ID "(" args ")" "timeout" dur "else" expr
       | "future" target "." ID "(" args ")" "deadline" dur
       | "await" expr "timeout" dur "else" expr
       | "match" expr "{" ("case" pat "->" expr)+ "}"
       | "(" expr ")"
binop ::= "+" | "-" | "*" | "/" | "div" | "mod" | "++"
        | "==" | "!=" | "<" | ">" | "<=" | ">=" | "&&" | "||"
target ::= ID | "self" | "remote" "(" STRING "," STRING ")" ":" ID
```

要点を四つ挙げる。

- **now と await は期限と else を伴う。**期限なしの形は文法に存在しない（**P2**）。
- ==、!=、&&、||、! を入れる。現状これらは書けない。
- / は浮動小数の除算、div / mod は整数除算とする。int 同士の / が float を返す現状の曖昧さを解消する。
- リモート送信先は remote("host:port","actor") : IFace と interface を明記する。これで境界の any が消える。

20.4 文

```
stmt  ::= "var" ID ":" type "=" expr ";"
        | ID "=" expr ";"
        | expr ";"
        | "reply" "(" expr ")" ";"
        | "send" target "." ID "(" args ")" ";"
        | "express" "send" target "." ID "(" args ")" ";"
        | "if" "(" expr ")" stmt ("else" stmt)?
        | "for" ID "in" expr ".." expr "do" stmt          // 有界
        | "while" "(" expr ")" stmt                       // プロファイル A のみ
        | "atomic" block
        | "select" "{" case+ timeout? "}"
        | "become" ID "(" args ")" ";"
        | "match" expr "{" ("case" pat "->" stmt)+ "}"
```

```
        | block
case  ::= "case" ID "(" params ")" guard? "->" block
```

`for` は上下限が明らかな有界ループで、両プロファイルで使える。`while` はプロファイル A のみとする。

21　型システム

21.1　判断の形

型付けは**型と効果の対**で述べる。

$$\Gamma \vdash_{C.m} e : \tau \,!\, \varepsilon$$

Γ は変数環境、$C.m$ は検査中のメソッド（`reply` の帰属先）、τ が型、ε が効果集合である。

21.2　主要な規則

$$\frac{ret(C,m) = \tau \qquad \Gamma \vdash_{C.m} e : \tau \,!\, \varepsilon}{\Gamma \vdash_{C.m} \texttt{reply}(e) : \texttt{unit} \,!\, \varepsilon}\ \textsc{Reply}$$

$$\frac{\Gamma \vdash a : \texttt{actor}(C') \qquad C'.m' : (\vec{\tau}) \to \tau' \,!\, \varepsilon' \qquad \Gamma \vdash e_i : \tau_i \,!\, \varepsilon_i}{\Gamma \vdash \texttt{send}\ a.m'(\vec{e}) : \texttt{unit} \,!\, \bigcup \varepsilon_i}\ \textsc{Send}$$

$$\frac{\Gamma \vdash a : \texttt{actor}(C') \qquad C'.m' : (\vec{\tau}) \to \tau' \,!\, \varepsilon' \qquad \Gamma \vdash e_i : \tau_i \,!\, \varepsilon_i \qquad \Gamma \vdash e_{alt} : \tau' \,!\, \varepsilon_a}{\Gamma \vdash \texttt{now}\ a.m'(\vec{e})\ \texttt{timeout}\ d\ \texttt{else}\ e_{alt} : \tau' \,!\, \varepsilon' \cup \varepsilon_a \cup \bigcup \varepsilon_i}\ \textsc{Now}$$

$$\frac{\Gamma, \overline{x{:}\tau} \vdash g : \texttt{bool} \,!\, \emptyset \qquad \Gamma, \overline{x{:}\tau} \vdash_{C.m} body : \texttt{unit} \,!\, \varepsilon_b \qquad \varepsilon_b \subseteq \varepsilon \qquad replies(body) = 1}{\Gamma \vdash \texttt{method}\ m(\overline{x{:}\tau}) : \tau_r \,!\varepsilon\ \texttt{where}\ g\ \{body\}}\ \textsc{Method}$$

三点が重要である。

1. **Send は callee の効果 ε' を伝播させない。**効果は別アクターで起こるからである。
2. **Now は伝播させる。**呼び出し側が待つので、待っている間の性質は呼び出し側の責任になる。**この一点だけで境界の方向の規律が導かれる**—- プロファイル R は `net` や `ai` を持てないので、それらを持つメソッドを `now` で呼べない。
3. **ガードは効果を持てない**（$\varepsilon = \emptyset$）。受理の試行で状態が変わってはならない。

21.3　返信の規律

$replies(body) = 1$ は「全実行パスでちょうど一度 `reply` する」である。現在は上界を構文で、下界を注釈時にのみ検査しているが、戻り値型が必須になれば両方を常に課せる。

`select` の case 本体と express メソッドの本体は**別の**返信文脈である。それぞれ対応するメッセージの ret に照合される。

21.4 物理単位

単位は型の添字であり、乗除で指数を足し引きする。

$$\frac{}{\texttt{float}\langle u_1\rangle \times \texttt{float}\langle u_2\rangle \to \texttt{float}\langle u_1 u_2\rangle} \qquad \frac{}{\texttt{float}\langle u\rangle + \texttt{float}\langle u\rangle \to \texttt{float}\langle u\rangle}$$

加減は単位が一致していなければ型エラーとする。依存型は不要である。

22 効果

効果	意味	対応する既存 capability
`mut`	自分の状態を書き換える	—
`time`	時刻の取得、待機	`Time`
`io`	宣言した装置への入出力	`UI.SDL`、GPIO 等
`mem`	動的割り付け（`new`、配列の伸長）	`Core.Array`、`Actor`
`net`	ノード外への通信	`Web`、`AIOS.Remote`
`ai`	モデル推論	`AIOS.Model`（機外へ出るものは `net` も併記）
`fs`	永続化	`AIOS.Memory`
`log`	観測のみの出力	`Console`、`AIOS.Event`

対応表があるので、既存の 17 分類から機械的に初期値を与えられる。新しく設計する必要はない。

注意 2. `ai` と `net` を分けるのが要点である。オンデバイス推論は `{ai}` だけを持ち `net` を持たない。これにより「AI を使うが機外へは出ない」がシグネチャに現れる。フィジカル AI では、この区別が安全性と秘匿の両方で意味を持つ。

23 プロファイル

	`agent`	`realtime`	`boundary`
許す効果	すべて	`mut`, `time`, `io`, `log`	`mut` のみ
`new`	可	初期化フェーズのみ	不可
配列	可変長可	固定長のみ	固定長のみ
`while`	可	不可（`for` のみ）	不可
再帰・自己送信の連鎖	可	不可	不可
`now` / `await`	期限必須	期限必須。かつ相手の効果が許容集合に収まる場合のみ	不可
浮動小数	可	`fix<i,f>` または単位つき `float` のみ	不可
`timing` 宣言	任意	**必須**	—
`mailbox` 宣言	任意	**必須**	必須
`express`	可	可（ハンドラも R の制約に従う）	可

プロファイルは**型システムの外側の構文・グラフ解析**で強制する。依存型は要らない。そして Now の効果伝播規則があるので、「実時間アクターがエージェントを同期で呼ぶ」は**効果の検査だけで**禁止さ

れる。

24　実行モデル

24.1　アクターの状態

各アクターは通常キューと express キューを持つ。

状態	内容
idle	受理可能なメッセージを待っている
running	メソッド本体を実行中
blocked	期限付きで返信を待っている（期限で必ず解ける）
atomic	`atomic` 区間の実行中。express を保留する

ABCL/1 の dormant / active / waiting に対応する。違いは blocked が必ず有限時間で解けることである（**P2**）。

24.2　受理規則 —- 一本の経路

1. express キューに、名前が一致し**ガードを持たない** express メソッドに対応するメッセージがあれば、それを受理する。atomic 中は保留する。
2. そうでなければ、通常キューを到着順に走査し、**名前が一致し、かつガードが真**である最初のメッセージを受理する。
3. どちらも無ければ idle で待つ。新しい到着、または状態の変化があれば再走査する。

`select` は「対象を指定した名前集合に限り、期限を付けた」規則 2 の特別な場合である。これで受信経路が一本になり、現在の先取り競合（§3 の `select` の注意）が消える。

24.3　`atomic`

- `atomic` 区間では express を受理しない。
- `atomic` 区間では `now` / `await` を書けない（構文検査）。待ちながら割り込みを止めることを許さない。
- `atomic` 区間は有界でなければならない（プロファイル R では `for` のみ）。

24.4　返信

`reply` は、いま受理しているメッセージに一度だけ答える。express ハンドラや `select` の case で受理したメッセージはそれぞれ独立の返信先を持つ。

25　現状との差分

25.1　足すもの

項目	効く目的
`== != && || !`	両方
`record` / `variant` と `match`	主にエージェント
`interface` 宣言とリモートの型付け	両方
効果注釈と検査	両方（プロファイルの土台）
`profile` 宣言	両方
`where` ガード	両方
`express` と `atomic`	主に実時間
`mailbox` 容量と溢れ方針	主に実時間
`timing`（周期・期限・優先度）	実時間
`fix<i,f>` と物理単位	実時間
`for` 有界ループ	実時間
メソッド呼び出しフレーム（実装）	express の前提

25.2　変えるもの

	現状	理想仕様
戻り値型	省略可（推論）	**必須**。推論は補助
仮引数・フィールド	型を書かない	型を書く
`now` / `await`	期限なし	**期限と `else` 必須**
`/`	`int` 同士でも `float`	`/` は浮動小数、`div`/`mod` が整数
`sender`	`any`、送信は無検査	`Reply<`ρ`>`。返信のみ
リモート送信先	`any`	`interface` 型
受信経路	名前 dispatch と `select` が並立	一本（`select` は特別な場合）
`become`	引数のみ検査	インタフェースの一致を検査
アクター型	単一化で区別されない	クラス名で区別

25.3　禁止するもの（プロファイル R）

期限なしの待ち、`while`、再帰と自己送信の連鎖、初期化後の `new` と配列の伸長、浮動小数（単位つきと固定小数点を除く）、`net` / `ai` / `fs` 効果、`atomic` 中の待ち。

26　この仕様で言えるようになること

性質	現状	理想仕様
型健全性（保存・進行）	言える	言える（効果つきに拡張）
返信がちょうど一度	注釈時のみ	**常に言える**
境界の型	`any`	**言える**（interface）
デッドロック自由	一般には言えない	**期限付き待ちのみなので、有限時間で解ける**
停止性	言えない	**プロファイル `R` では言える**
応答時間の上界	言えない	**プロファイル `R` では解析できる**（宣言があるので）
機外へ出ないこと	言えない	**効果で言える**
単位の整合	言えない	**言える**
公平性	言えない	言えない（`where` で悪化する）

現状「言えない」とされていた四つ —- デッドロック自由、停止性、応答時間、外部流出 —- が、**プロファイルを宣言させることで言えるようになる**。これが第 II 部の結論の具体的な中身である。

引き換えに公平性は悪化する。`where` を入れるとガードが永遠に真にならないメッセージが残りうるからで、これは仕様として明記し、診断で補うべき部分である。

27　最小の到達点

全部を一度に作る必要はない。次の四つが揃った時点で「二つの目的に耐える言語」と言える。

1. **効果注釈と検査**（既存 capability 分類の静的化）。プロファイル分離、境界の方向の規律、外部流出の保証がここに乗る。
2. **期限付き `now` / `await` のみ**。デッドロック自由が言えるようになる。
3. **`interface` 宣言**。`any` が消える。
4. **`profile realtime` の有界性検査**。停止性と応答時間が言えるようになる。

`record` / `variant`、物理単位、`where`、express mode は、この四つの上に順に足していける。

参考文献

[1] Akinori Yonezawa, Jean-Pierre Briot, Etsuya Shibayama. Object-Oriented Concurrent Programming in ABCL/1. *OOPSLA 1986*, pp. 258–268.

[2] Akinori Yonezawa (ed.). *ABCL: An Object-Oriented Concurrent System*. MIT Press, 1990.

[3] 本リポジトリ `docs/aipl_vs_abcl1_ja.pdf` — AIPL と ABCL/1 の機能・モデル比較。

[4] 本リポジトリ `docs/aipl_soundness_ja.pdf` — AIPL^{-} の型健全性・型安全性の Coq 検証、哲学者の食事問題、および「保証しないこと」の一覧。

[5] 本リポジトリ `docs/aipl_reply_inference_ja.pdf` — `reply` からの戻り値型推論と `select` における返信の帰属の扱い。

[6] 本リポジトリ `docs/aipl_guide_ja.pdf` — 現状の言語ガイド。

[7] Alan Burns, Brian Dobbing, Tullio Vardanega. Guide for the use of the Ada Ravenscar Profile in high integrity systems. *Ada Letters* XXIV(2), 2004.（同じ言語の中に検証可能な部分集合を切

るという設計の先例）

[8] Mark Miller, E. Dean Tribble, Jonathan Shapiro. Concurrency Among Strangers: Programming in E as Plan Coordination. *TGC 2005.*（ブロックを一切許さず eventual send と promise だけで書かせる設計）

[9] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, Daniel Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE* 79(9), 1991.（実時間記述における有界性と時間の宣言）