AICE: 型付き Actor に基づく知的計算環境

AIPL 記述言語、型の必要性、AIOS 実行基盤の考察

Yasushi Kodama Repository Report

2026 年 5 月 4 日

概要

AICE（Actor-based Intelligent Computing Environment）は、AI エージェント、モデル呼び出し、記憶、ツール、UI、Web endpoint、OS service を actor として統一的に扱う知的計算環境である。本稿は、AICE の基本構想を、記述言語 AIPL（現在の ABCL/c+ 実装を基礎とする actor-first 言語）との関係から整理する。

AICE の中心的な主張は、AI アプリケーションを単一の巨大な関数や逐次スクリプトとしてではなく、型付き message を交換する多数の actor の協調として捉える点にある。この見方により、planner、solver、reviewer、memory、model adapter、tool adapter などを同じ実行モデルで接続できる。一方で、AI system は外部 API、非決定的なモデル出力、並行実行、remote actor、長時間処理を含むため、単純な動的 dispatch だけでは安全性と保守性を確保できない。

本稿では、AICE において型が必要となる理由を、値の整合性、message payload の整合性、actor capability の明示、future/await の結果管理、workflow protocol の順序保証という観点から論じる。さらに、通常の型安全性だけでは非同期通信の正しさを保証できないことを示し、session type または runtime session protocol checker を併用する設計を提案する。

目次

1 はじめに 3

2 AICE の目的 3

3 AIPL の位置づけ 4

4 AICE のアーキテクチャ 5

5 なぜ型が必要か 5

5.1 値の整合性 6

5.2 message payload の整合性 6

5.3 capability の明示 6

5.4 future/await の結果管理 7

5.5 workflow protocol の保証 7

6 型安全性だけでは足りない理由 7
6.1 順序の問題 7
6.2 非決定性の問題 7
6.3 外部境界の問題 7

7 AIPL の型システム 8
7.1 値型 8
7.2 actor 型 8
7.3 通信型 8

8 AICE における session type 8

9 AICE の典型的 workflow 9

10 AICE と従来方式の比較 10

11 限界と課題 10
11.1 AI 出力の型 10
11.2 remote actor 11
11.3 deadlock 11
11.4 any の制御 11

12 研究課題としての AICE 11

13 結論 11

1　はじめに

近年の AI アプリケーションは、単一のモデル呼び出しだけでは完結しない。典型的なシステムは、ユーザ入力を受ける UI、問題を分解する planner、解答を生成する solver、出力を検査する reviewer、履歴を保持する memory、外部サービスを呼ぶ tool adapter、複数の LLM provider を切り替える model router から構成される。

この構成は自然に並行的であり、分散的であり、失敗を含む。model call は遅延し、network は失敗し、tool output は予期しない形になり、AI の出力は非決定的である。従来の逐次スクリプト型の設計では、これらの構成要素を結合する glue code が肥大化し、どの component がどの message を受け取り、どの順序で処理すべきかが暗黙化しやすい。

AICE はこの問題に対して、actor を第一級の実行単位とする設計を採る。AICE において、OS service、AI agent、memory、model adapter、tool adapter、UI、Web endpoint はすべて actor である。actor は内部状態を閉じ込め、外部とは message によってのみ相互作用する。AIPL は、この AICE を記述するための型付き actor 言語である。

■本稿の貢献

- AICE の計算モデルを actor-first AIOS として整理する。
- AIPL の役割を、AICE を記述するための言語として説明する。
- AIOS において型が必要となる理由を、値・message・capability・future・protocol の観点から分析する。
- 通常の型安全性だけでは非同期通信の正しさを保証できないことを論じる。
- session type または runtime session protocol checker による補完を提案する。

2　AICE の目的

AICE は Actor-based Intelligent Computing Environment の略であり、知的処理を actor 群の協調として実行する環境である。ここでいう知的処理とは、LLM 呼び出しだけを意味しない。問題分解、記憶参照、ツール実行、レビュー、ユーザ応答、外部 API 連携を含む、AI application 全体の実行過程を指す。

AICE の目的は次の三つに整理できる。

1. AI component を actor として統一的に扱うこと。
2. actor 間 message を型で検査し、接続ミスを早期に発見すること。
3. 複数 actor にまたがる workflow を protocol として管理すること。

第一の目的は、AIOS の構成要素をすべて同じ抽象で扱うためである。planner も memory も model adapter も、外部から見れば method を持つ actor である。内部実装が LLM call であっても、database access であっても、UI update であっても、AICE の接続面は message passing に統一される。

第二の目的は、AI system の glue code を安全にするためである。たとえば `solver.solve(plan)` が文字列 plan を要求しているのに、planner が配列や数値を送ってしまうと、エラーは workflow の中盤で露出する。型付き message は、この種の shape mismatch を実行前に発見する。

第三の目的は、局所的な message の型だけでは足りないためである。planner、solver、reviewer の各 method が正しい型を持っていても、reviewer を solver より先に呼べば workflow としては壊れる。AICE は actor の局所型と、workflow の大域的 protocol の両方を必要とする。

3 AIPL の位置づけ

AIPL は AICE を記述するための actor-first 言語である。現在の実装では ABCL/c+ がその基礎であり、`class`、`method`、`new`、`send`、`now`、`future`、`await`、`select`、`become`、`remote` といった構文を持つ。

AIPL の基本的な考え方は、class を単なる object の blueprint ではなく、actor の behavior 定義と見ることである。

```
class Planner {
  method plan(problem) {
    reply("decompose: " + problem);
  }
}

class Solver {
  method solve(plan) {
    reply("answer for " + plan);
  }
}

var planner = new Planner();
var solver = new Solver();

var plan = now planner.plan("question");
var answer_future = future solver.solve(plan);
var answer = await answer_future;
```

この例では、`Planner` と `Solver` はそれぞれ独立した actor として生成される。`now` は同期的に結果を待つ send、`future` は非同期に結果 handle を返す send、`await` は future の結果を取り出す構文である。

AIPL は、AICE に対して次の役割を果たす。

- actor の interface を method として記述する。
- actor 間の接続を `send`/`now`/`future` として明示する。
- actor 内部状態を actor の field として閉じ込める。
- AIOS primitive を capability として呼び出す。
- workflow protocol を session として記録・検査する。

4　AICE のアーキテクチャ

AICE は、言語、runtime、capability、protocol layer からなる。

表 1: AICE の層構造

層	役割
AIPL language	actor、message、future、select、protocol を記述する表層言語
Type system	値、関数、array、future、actor method、capability の整合性を検査する
Actor runtime	actor table、mailbox、scheduler、method dispatch、reply/future を管理する
Capability layer	UI、Web、Memory、Model、Tool、Network など外部資源への明示的入口
Protocol layer	複数 actor にまたがる workflow の順序を検査する

AICE では、AI application の component は表 2 のような actor として表現される。

表 2: AICE における典型的 actor

actor	役割
UserActor	ユーザ入力を受け取り、結果を返す
PlannerActor	問題を subtask に分解する
SolverActor	plan に基づき解答を生成する
ReviewerActor	解答を検査し、改善点を返す
MemoryActor	会話履歴、事実、作業状態を保存・検索する
ModelActor	LLM provider を呼び出す
ToolActor	shell、HTTP、database、file など外部 tool を実行する
RouterActor	model や tool の選択を行う
WebActor	Web endpoint と actor message を接続する

5　なぜ型が必要か

AICE において型が必要な理由は、単に「整数と文字列を間違えない」ためではない。AIOS では、型は component 間接続の設計情報であり、capability の境界であり、workflow の検査可能性を作る基盤である。

5.1 値の整合性

最も基本的な型の役割は、値の形を保証することである。数値計算 primitive は数値を受け取り、文字列処理 primitive は文字列を受け取る。AIPL の型には `int`、`float`、`string`、`bool`、`unit`、array、future がある。

AI system では、文字列化された JSON、自然言語、数値 score、tool result が混在する。すべてを文字列として扱う設計は短期的には簡単だが、長期的には接続ミスを隠す。型は「この値は prompt なのか、model response なのか、task id なのか、score なのか」を区別するための最初の構造である。

5.2 message payload の整合性

actor 間通信では、method 名だけでなく payload の shape が重要である。

$$\texttt{solver.solve} : \texttt{Plan} \to \texttt{Answer}$$

のような interface があるとき、planner が `Plan` を返し、solver が `Plan` を受け取ることを検査できる。現在の ABCL/c+ 実装では method return type はまだ限定的だが、actor method registry により、少なくとも local actor への send について method の存在、arity、引数型を検査できる。

これは dynamic actor system との大きな違いである。動的 dispatch では、`send solver.solv(...)` のような typo や、引数数の誤りが runtime まで残る。AICE では、このような接続ミスを AIPL の型検査で早期に落とす。

5.3 capability の明示

AICE では、外部資源へのアクセスを capability として扱う。たとえば Model、Memory、Tool、Network、Web、UI は capability である。型は、どの actor がどの capability を要求し、どの形の値をやり取りするかを明示する。

たとえば model call は次のような型を持つべきである。

$$\texttt{model.generate} : \texttt{Prompt} \to \texttt{future Response}$$

memory は次のような型を持つ。

$$\texttt{memory.put} : \texttt{Key} * \texttt{Value} \to \texttt{unit}$$

このように capability に型を与えると、外部 API の呼び出しは無秩序な副作用ではなく、検査可能な境界になる。

5.4 future/await の結果管理

AI model call や remote tool call は遅い。したがって AICE では future が重要になる。future に型がなければ、非同期処理の結果が何であるかは await する時点まで曖昧になる。

$$\frac{\Gamma \vdash e : \texttt{future}\ \tau}{\Gamma \vdash \texttt{await}\ e : \tau}$$

この規則により、非同期処理であっても結果型は失われない。複数の AI actor に同時問い合わせを行う場合でも、どの future がどの結果型を持つかを静的に管理できる。

5.5 workflow protocol の保証

通常の型は、個々の message が正しい shape を持つことを保証する。しかし、message の順序までは保証しない。次の二つの message がどちらも well-typed であっても、順序が逆なら workflow としては誤りである。

```
future solver.solve(plan);
now reviewer.review(answer);
```

reviewer は solver の answer を必要とする。したがって AICE には、値の型に加えて通信 protocol の型が必要になる。これは session type の役割である。

6 型安全性だけでは足りない理由

通常の型安全性は「well-typed program は型エラーで stuck しない」という性質である。しかし actor system では、それだけでは十分でない。

6.1 順序の問題

actor A が actor B に正しい型の message を送り、actor C にも正しい型の message を送ったとしても、その順序が protocol に反していれば system は誤った結果を出す。型安全性は局所的な message の shape を保証するが、workflow の時系列までは保証しない。

6.2 非決定性の問題

mailbox には複数の message が到着する。selective receive を持つ actor は、どの message を先に取り出すかによって動作が変わる。message が well-typed であっても、選択順序が仕様と異なると意味的に誤る。

6.3 外部境界の問題

LLM、HTTP API、file system、shell command は完全には静的に検査できない。AICE はこの境界を `any` や dynamic contract として扱う必要がある。重要なのは、動的境界を隠すので

はなく、型と capability によって明示することである。

7 AIPL の型システム

AIPL の型システムは、値型、actor 型、通信型の三層として設計できる。

7.1 値型

$$\tau ::= \texttt{int} \mid \texttt{float} \mid \texttt{string} \mid \texttt{bool} \mid \texttt{unit} \mid \tau[] \mid \texttt{future}\ \tau \mid \texttt{any}$$

any は dynamic boundary のための型である。AICE では any を完全に排除するのではなく、remote actor、AI model output、JSON、外部 tool result などに限定して使うべきである。

7.2 actor 型

actor 型は method signature を含む。

$$\texttt{actor}(C)\{m_1 : \tau_1; \ldots; m_n : \tau_n\}$$

たとえば planner は次のような型を持つ。

$$\texttt{actor}(\mathit{Planner})\{\texttt{plan} : \texttt{Problem} \to \texttt{Plan}\}$$

actor 型により、send target が本当にその method を持つか、引数型が合っているかを検査できる。

7.3 通信型

通信型は、単一の message ではなく message 列を扱う。

$$P ::= a.m;\ P \mid \mathit{end}$$

たとえば、典型的な solve workflow は次である。

$$\mathit{planner.plan};\ \mathit{solver.solve};\ \mathit{reviewer.review};\ \mathit{end}$$

現在の実装では、この protocol は runtime checker として実装されている。将来的には AIPL の静的 session type として型検査器に組み込むべきである。

8 AICE における session type

AICE に session type が必要な理由は、AI workflow が複数 actor にまたがるからである。planner、solver、reviewer はそれぞれ独立に正しい actor でも、呼び出し順序が壊れれば system 全体は正しくない。

session type は、次に許される message を型環境に保持する。

$$\Gamma; \Delta \vdash s \triangleright \Delta'$$

ここで Γ は通常の型環境、Δ は session environment である。

$$\frac{\Delta(s) = a.m : \tau_1 \to \tau_2; P \quad \Gamma \vdash e : \tau_1}{\Gamma; \Delta \vdash \texttt{now}\ a.m(e) : \tau_2 \triangleright \Delta[s := P]}$$

この規則は、`a.m` が現在の session で次に許された action のときだけ `now` を許し、実行後に protocol を tail へ進める。

runtime checker でも同様の考え方は実現できる。現在の ABCL/c+ 実装では `protocol_define`、`protocol_start`、`protocol_state`、`protocol_end` により、actor.method の線形列を実行時に検査する。

```
protocol_define("SolveProtocol",
  "planner.plan -> solver.solve -> reviewer.review");
var sid = protocol_start("SolveProtocol");
```

この runtime 検査は、完全な静的 session type ではないが、AICE の workflow safety に向けた現実的な第一段階である。

9 AICE の典型的 workflow

AICE での問題解決 workflow は、次のように書ける。

```
class Planner {
  method plan(problem) {
    reply("plan for " + problem);
  }
}

class Solver {
  method solve(plan) {
    reply("answer based on " + plan);
  }
}

class Reviewer {
  method review(answer) {
    reply("OK: " + answer);
  }
}

var planner = new Planner();
var solver = new Solver();
var reviewer = new Reviewer();
```

```
protocol_define("Solve",
  "planner.plan -> solver.solve -> reviewer.review");
var sid = protocol_start("Solve");

var plan = now planner.plan("larger problem");
var f = future solver.solve(plan);
var answer = await f;
var verdict = now reviewer.review(answer);
protocol_end(sid);
```

この program では、planner、solver、reviewer は actor として分離される。solver は future で非同期に実行され、reviewer は answer を受け取った後に呼ばれる。protocol は呼び出し順序を検査する。型は payload の shape を検査する。

10 AICE と従来方式の比較

表 3: AICE と逐次スクリプト方式の比較

観点	逐次スクリプト方式	AICE
構成単位	関数、手続き、API call	actor
状態管理	global 変数や shared object に寄りやすい	actor 内に閉じる
並行性	明示的 thread/future 管理が散在	actor message と future に統一
接続検査	runtime error になりやすい	message payload を型検査
外部資源	ad hoc な API call	capability として明示
workflow 順序	コメントや convention に依存	protocol/session で検査
AI component	特別扱いされやすい	model/memory/tool も actor

AICE の利点は、AI system の構成要素を同じ抽象で扱える点にある。すべてを actor にすることは、単なる実装上の統一ではない。それは、型、capability、protocol を同じ message interface に適用できるという設計上の統一である。

11 限界と課題

AICE と AIPL には未解決課題もある。

11.1 AI 出力の型

LLM は自然言語を返す。自然言語は構造化されていないため、完全な静的型付けは難しい。AICE では、model output を `string` または `any` として受け取り、parser actor や validator

actor によって構造化型へ変換する設計が必要である。

11.2 remote actor

remote actor は local method registry で完全には検査できない。interface definition を共有する、または remote boundary に contract を置く必要がある。

11.3 deadlock

`now` は同期 send であるため、actor 間で循環的に `now` を呼ぶと deadlock が起こり得る。型だけで deadlock freedom を保証するには、effect type、session type、または capability policy が必要になる。

11.4 `any` の制御

`any` は実用上必要だが、広く使いすぎると型の意味を失わせる。AICE では `any` を dynamic boundary に閉じ込め、境界から内側へ入る時点で validation を行うべきである。

12 研究課題としての AICE

AICE は、実装システムであると同時に、研究課題でもある。特に重要な問いは次である。

1. actor method return type と `reply` をどのように型で結ぶか。
2. future 型と AI model call の失敗をどう表現するか。
3. session type を AIPL の Hindley-Milner 風推論とどう共存させるか。
4. remote actor interface をどう配布・検証するか。
5. LLM の自然言語出力をどの段階で構造化型へ変換するか。
6. capability policy を型システムへ入れるか、runtime 権限として扱うか。

これらは、AIOS の安全性を考える上で避けられない。特に重要なのは、「AI の非決定性を型で消す」のではなく、「非決定的な部分を境界として明示し、その周囲の orchestration を型で守る」という立場である。

13 結論

AICE は、AI application を actor 群の協調として捉える知的計算環境である。AIPL はそのための記述言語であり、actor、message、future、select、protocol を言語要素として扱う。

AICE に型が必要な理由は、単なる値の分類に留まらない。型は actor interface を明示し、message payload を検査し、capability 境界を可視化し、future の結果を追跡し、session protocol の静的化へ進むための土台になる。

一方で、通常の型安全性だけでは、非同期 actor system の通信順序までは保証できない。したがって AICE は、値と message の型に加え、session type または runtime protocol checker を

必要とする。この二層構造、すなわち局所的な型安全性と大域的な workflow 安全性の組み合わせが、AICE の中心的な設計原理である。