通信Actor自动机——将Erlang进程建模为通信机器

DominicOrchard Mihail Munteanu Paulo Torrens英国肯特大学 Masabi有限公司
英国肯特大学 d.a.orchard@kent.ac.uk mihailmunteanu944@gmail.com
paulotorrens@gnu.org

Brand和Zafiropulo提出的通信有限状态机（通信有限状态机）概念，以信道为基础，为消息传递并发提供了一种简洁而强大的模型。然而，消息传递并发的一种重要变体无法由通信有限状态机直接表示：Actor模型。在本文中，我们定义了通信有限状态机的一种变体，称为通信参与者自动机，用于表示Erlang所提供的Actor模型并发：该模型具有邮箱，消息按照反复应用模式匹配的方式从邮箱中接收。此外，这种通信有限状态机变体支持动态进程拓扑，从而捕获基于Actor的消息传递并发场景中的常见编程习惯。这为Erlang程序的建模、规约和验证提供了新的基础。我们还研究了一类能够避免竞争条件的CAA。

1 引言

现代软件开发往往不再遵循传统的顺序计算方式，而是以并发为核心：编写代码时，允许多个进程同时运行，并可能共享资源。随着软件系统日益复杂，人们开发了各种平台和编程语言，以促进并发、并行和分布式计算，例如 Erlang 编程语言 [1, 2]。对这类系统进行形式化规约的需求，促使人们设计出能够对程序进行推理的形式系统。

Brand和Zafiropulo提出的通信有限状态机（CFSM，也称为通信自动机）为描述并发且相互通信的进程提供了一种模型，并可据此描述良构的通信协议 [7]。其核心思想是将一组（有限个）并发进程建模为有限自动机（每个进程对应一个自动机），其标签对应于在连接每对自动机的信道上发送和接收消息；这些信道统称为一个协议。借助这类精确描述，可以研究通信进程的性质，例如检查每条消息是否都能被接收，或者是否有进程一直等待一条永远不会发送的消息。

该模型有助于进一步研究并发系统各种性质的可判定性或不可判定性（例如，[16, 17, 11, 13, 14]）。例如，Brand和Zafiropulo证明，当系统仅限于两台机器且消息类型只有一种时，有界性（即通信可以在有界队列中进行）、无死锁性和未指定接收（发送了却未被接收的消息）都是可判定的性质 [7]。此外，判定这些性质可以在多项式时间内完成 [18], ，而且即使仅将一台机器限制为只有一种消息类型，也可以进行判定。近年来，通信有限状态机还被用作核心建模工具，为其他并发程序模型（如图形化编排 [15]）提供了有用的接口。

在 CFSM 模型中，进程通过连接各个进程的信道（先进先出队列）进行通信。因此，进程知道消息是从哪个其他进程接收的。这与参与者

Ilaria Castellani 和 Alceste Scalas（编）：第14届编程语言并发与通信中心软件方法研讨会（PLACES2023）。EPTCS 378，2023年，第38–48页，doi:10.4204/EPTCS.378.4

© D. Orchard、M. Munteanu 和 P. Torrens

模型不同，在参与者模型中，每个进程都有一个“邮箱”，其他进程会将消息投递到其中，但不一定会附带任何发送者信息。这使得传统的通信有限状态机难以描述基于参与者的方法。通信有限状态机的另一个局限是，它们只能描述通信拓扑固定的程序：模型中的消息发送方和接收方都是固定的，进程无法动态地将消息发送给不同的进程。然而，这并非并发编程场景中的主流编程惯用方式。通信有限状态机还规定了简单的消息接收模型，无法描述更细粒度的接收方法，例如 Erlang 的邮箱语义；这种语义允许处理并非最新收到的消息，并利用语言层面的模式匹配。即便如此，通信有限状态机与 Erlang 的计算模型仍有惊人的相似之处：每一对进程都代表能够双向通信的顺序计算。

我们提出一种 CFSM 变体，用于描述 Erlang 的异步邮箱语义；此外，通过在进程的自动机中引入记忆机制，允许对进程标识符的变量进行绑定（和重新绑定），从而支持动态拓扑结构。第 2 节阐述该模型，并给出与简单 Erlang程序对应的模型示例。第 3 节探讨此类模型的性质。第 4 节通过讨论相关工作和后续步骤作结。

1.1 通信有限状态机

为便于比较，我们简要回顾 CFSMs [7] 的形式化定义。由 N 个 CFSM 组成的系统称为协议，它包含四个组成部分（通常表示为一个四元组）：

- $(S_i)_{Ni=1}$ 是 N 个（互不相交的）有限集合 S_i，给出每个进程 i 的状态集合；
- $(o_i)_{Ni=1}$ 其中 $o_i \in S_i$ 是每个进程 i 的初始状态；
- $(M_{i,j})_{Ni,j=1}$ 是 N 个（两两不相交的）有限集合，其中 $M_{i,j}$ 表示可从进程 i 发送到进程 j 的消息集合，而当 $i = j$ 时，$M_{i,j} = /0$；
- $(\text{succ} : (S_i \times \bigcup_{Nj=1}(M_{i,j} \uplus M_{j,i})) \to S_i)_{Ni=1}$ 是 N 个状态转移函数（偏函数），其中 succ(s,l) 根据状态 s 以及给定消息 l 计算进程 i 的后继状态 s'；该消息要么是从 i 发送到 j 的（因此 $l \in M_{i,j}$），要么是 i 从 j 接收的（因此 $l \in M_{j,i}$）。

典型的表示方式将上述带索引的集合视为有限序列。我们用符号 l 表示消息，因为后文会将它们称为自动机的“标签”。

对于一个协议，全局状态（或配置）是一个有序对 (S,C)，其中 S 是每个进程对应的状态序列 $S = \langle s_1 \in S1, \ldots s_N \in S_N \rangle$，而 C 是一个 $N \times N$ 矩阵，其元素 $c_{i,j} \in C$ 是从 $M_{i,j}$ 中取出的有限序列，表示进程 i 和 j 之间的 FIFO 消息队列（通道）（此处以及下文中，我们使用 $\cdot$ 连接此类序列，并用 $[l1, \ldots, l_m]$ 表示一个包含 m 条消息的序列实例）。

二元关系 step描述了一个全局状态 (S,C)何时可以由于单个 succ 函数而演化为另一个全局状态 (S',C')。也就是说，$(S',C') \in \text{step}(S,C)$ 当且仅当存在 i、k、$l_{i,k}$ 并且满足以下任一条件：

$$\begin{array}{rll} 1. & (i \text{ sends to } k): & s'_i \in S' = \text{succ}_i(s_i, l_{i,k}) \text{ and } c'_{i,k} \in C' = c_{i,k} \cdot [l_{i,k}] \\ \textbf{or}\ 2. & (\text{reception from } k \text{ by } i): & s'_k \in S' = \text{succ}_k(s_k, l_{i,k}) \text{ and } c_{i,k} \in C = [l_{i,k}] \cdot c'_{i,k} \end{array}$$

我们采用 类似的术语和结构，但要有足够的变化，以体现 Erlang 的 Actor模型 g.

2 Erlang 通信有限状态机的一种变体

我们的主要目标是借鉴 Erlang的邮箱语义，定义一种CFSM 变体，我们称之为通信参与者自动机（CAA）。我们首先回顾一些 Core Erlang 的核心概念，随后

描述 CAA 及其组合形成协议。传统的通信有限状态机定义直接考虑若干进程的全局配置（状态），而我们将每个进程（代表一个 Erlang 参与者）视为独立的状态机，从而得到局部的“孤立”刻画，并由此推导出全局的“协议”刻画。

2.1 Erlang 基本定义

Erlang 的一个关键概念是邮箱：每个进程（也称为参与者）都有一个用于接收消息的队列，而不是严格按照先进先出（FIFO）的顺序处理消息；进程可以从队列中匹配消息：给定一系列模式，进程会按顺序选取队列中第一个与其中某个模式统一的消息。如果队列中的第一条消息与任何模式都不匹配，就会对下一条消息尝试所有模式，依此类推 [8]。下文将抽象地使用 Erlang 的项语法（详情参见 Carlsson 等人 [9]），具体采用哪种语法可能有所不同。具体语法以等宽字体表示，例如，{1, 2} 是由两个整数项组成的 Erlang 元组。

定义 2.1（语法类别）。令 Term 为项的集合，元素以 t 表示。令 Var $\subset$Term 为变量的集合，元素以大写字母表示。令 Proc $\subset$ Var 为进程标识符的集合，这些标识符唯一标识进程。令 Pat 为模式的集合，元素以 pat 表示。由于 Erlang 采用按值调用语义，我们定义项的子集 Value $\subset$Term，其中包含范式（称为值），元素以 v 表示。注意，Var $\subset$ Value。

需要指出的是，就本文提出的语义而言，我们主要关注四种基本项：进程标识符、变量、原子和元组。标识符和变量有助于控制拓扑结构，而原子和元组则有助于构造消息模式（从而帮助识别要发送或接收的消息）。在第 2.3 节将给出的示例中，我们还会考虑整数和算术运算符，不过这并非必要。项语法中也可以考虑函数，但本文不讨论这种可能性，因为我们旨在关注进程自动机本身的表达能力，而非项语言。

为了仿照 Erlang 接收消息的方式，我们需要一个合一的概念：传入的消息会与一组模式进行匹配，只有其中某个模式匹配成功时，进程才会继续执行。

定义 2.2（合一）。我们定义项与模式之间的合一 [8, 9]，记作 t $\triangleright$ 模式；合一要么产生 $\perp$，表示模式匹配失败，要么产生一个环境。该环境是从变量 V 的某个子集 $\subseteq$ Var 到项的有限映射，即 $\Gamma_V : V \to$ Term 表示成功模式匹配的绑定上下文。此类映射记作 γ。若 t $\triangleright$ 模式 $= \gamma$，且$\gamma \in \Gamma_V$对某个 V 成立，则将模式中的每个变量 v $\in$V 替换为 γ_V(v) 后，模式就等于 t。我们用 $\{X_1 \mapsto t1, \ldots, X_n \mapsto t_n\}$ 表示将 X_i 映射到 t_i 的环境（对所有 $1 \le i \le n$）。

2.2 单个进程的 CAA

与通信有限状态机一样，我们用有限状态自动机描述参与者。每个自动机都单独描述，代表一个拥有唯一标识符的 Erlang进程。我们的主要目的是描述 Erlang进程中的可能状态，即指出在执行过程的某个时刻，它可以接收和发送哪些消息，以及之后应该执行什么操作。

我们将参与者的概念正式定义如下（请注意，其中新增了终态，而通信有限状态机中并不总是包含终态）。

定义2.3。通信参与者自动机（CAA）是一个 7元组 (S, o, F $\mathscr{L}_1$, $\mathscr{L}_2$, δ, $<$)，其中包括一个有限状态集 S、一个初始状态 o $\in$S、一个可能为空的终态集 F $\subseteq$ S、一个集合

发送标签 $\mathscr{L}_!$ ⊆项、有限的接收标签集合 $\mathscr{L}_?$ ⊆模式、描述状态转移的函数 δ : S ×($\mathscr{L}_?$ ∪(变量 × $\mathscr{L}_!$)) →S，以及以 S 为索引的序关系族 <on $\mathscr{L}_?$。我们对 δ 的定义域施加限制，使得一个状态只能具有若干个接收标签或单个发送标签（但绝不能同时具有两者）。对于接收与 pat 匹配的消息的任何状态转移，我们记作 δ(s, ?pat)；对于发送消息的状态转移，我们记作δ(s, X !t)，其中 X !t 表示将由项 t 指定的消息发送到进程 X 这一对值 (X ,t)。对于状态 s ∈ S，我们将这类状态上的序关系记作 $<_s$。

请注意，CAA 本质上是一种确定有限自动机（DFA）状态机，其字母表定义为 $\Sigma = \mathscr{L}_? \cup$ (Var × $\mathscr{L}_!$)。协议中多个 CAA 的交互会呈现非确定性，相关内容将在第 2.3 节中介绍。因此，可以将“静态 CAA”（与移动 CAA 相对）理解为具有如下状态转移：S ×($\mathscr{L}_?$ ∪(Proc × $\mathscr{L}_!$)) → S，其中我们将与发送标签关联的变量替换为具体的进程标识符，即发送的目标始终是预先已知的。此处不再进一步探讨这一概念。

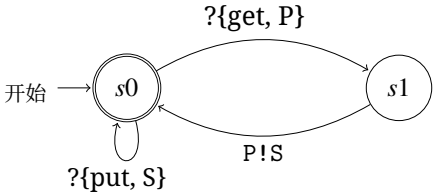
例 2.1。考虑以下 Erlang 代码，它定义了一个名为 mem 的函数，通过递归模拟一个内存单元：

```
mem（S） ->接收{get,P} ->P!S,mem(S);
{put,X} ->mem(X)结束 .
```

内存单元的状态由变量 S 表示。进程接收一对 {get, P}，随后将 S 发送给进程标识符 P；或者接收一对 {put, X}，随后以 X 为参数递归调用（即“更新后”的内存单元状态）。请注意，Erlang 中的小写项是原子。

启动后，这个函数可以建模为一个 CAA，其状态为 S = {s_0, s_1}，初始状态和终态为 o =s_0 and F = {s_0}，发送标签为 $\mathscr{L}_!$ =项，接收标签为 $\mathscr{L}_?$ = {{get, P}, {put, S}}，顺序为 {get, P} $<_{s_0}$ {put, S}，并具有以下转移（及对应的自动机）：

$$\begin{aligned} \delta(s_0, ?\{get, P\}) &= s_1 \\ \delta(s_0, ?\{put, S\}) &= s_0 \\ \delta(s_1, P!S) &= s_0 \end{aligned}$$



请注意，在上面的示例中，我们没有在“put”消息中使用变量 X，因为我们希望在递归调用中将接收到的值（即这对值的第二个分量）重新绑定到 S。我们不考虑 Erlang 语义的另一个方面：模式匹配中可能出现已绑定的变量，从而引发合一；这是一个额外的复杂问题，本文不予讨论。

上述定义足以刻画参与者的静态语义。然而，在执行期间，还需要更多信息来表示其在运行时的动态行为，即参与者的邮箱和内部状态。接下来，我们将对其进行形式化定义。

定义 2.4。局部状态（或机器配置）是一个三元组 (s, m γ)，由状态s、项的有限序列m以及环境 γ组成。项的序列m对未接收消息的消息队列（也称为参与者邮箱）进行建模，而γ 是参与者的内存。我们将 ε记作空序列，并将 [t1,...,t_n] 记作由 n个元素组成的序列，其中 t_1 是队列的头部。两个序列 m、m′可以拼接，记作 m ·m′，例如 $[1,2,3]\cdot[4,5] = [1,2,3,4,5]$。

2.3 CAA 系统：协议、状态和轨迹

与 CFSM 模型或 Erlang 程序一样，计算由并发参与者之间的通信来描述。为了对其进行形式化定义，我们通过状态之间的求值步关系，为多个 CAA 的组合（称为协议）赋予操作语义。

定义 2.5（协议）。协议是一个有限大小（或元数）为 N 的 CAA 索引族，写作$\langle(S_i, o_i, F_i, \mathscr{L}_{!i}, \mathscr{L}_{?i}, \delta_i)\rangle_{i=1}^{N}$。每个索引 i 都表示每个进程唯一的进程标识符。

定义 2.6（全局状态）。协议的全局状态（或系统配置）由 N 个局部状态组成的有限序列构成，写作 $\langle(s_i, m_i, \gamma_i)\rangle_{i=1}^{N}$ 其中每个 $s_i \in S_i$。我们将元数为 N 的协议的全局状态集合记为 G_N。

给定一个协议，我们可以导出所谓的初始全局状态：启动之前，每个进程的邮箱都是空的，并且处于其所属 CAA 中定义的初始状态。这个初始状态是确定性的：对于任何给定的协议，都只有一个可能的初始状态。

定义 2.7（初始全局状态）。对于一个协议，初始全局状态为 $\langle(o_i, \varepsilon, /0)\rangle_{i=1}^{N} \in G_N$，即每台机器都处于其初始状态，邮箱为空，并且变量到进程标识符的映射为空。

为了使用邮箱语义，我们定义一个偏函数，它仅在当前时刻可以接受某条消息时才有定义。与 Erlang 一样，我们按顺序检查邮箱中的每条消息；只有在当前没有任何可接受的模式匹配时，才尝试检查下一条消息。检查每条消息时，按定义的顺序检查模式，并且只接受第一个匹配的模式。

定义 2.8（pick 函数）。我们隐式假设一个 CAA $(S, o, F \ \mathscr{L}_!, \mathscr{L}_?, \delta, <)$ 作为上下文。于是，偏函数 $\text{pick}(s, m, v) = \langle \text{pat } \gamma \rangle$ 有定义，当且仅当：

- 对于所有模式′ $\in \mathscr{L}_?$，如果 $\delta(s, ?\text{模式})$ 已定义，那么对于所有 $v_k \in m$，我们有 $v_k \triangleright \text{模式} = \bot$；
- 对于所有模式′ $<_s$ 模式，我们有 $v \triangleright \text{模式}' = \bot$；并且
- 存在一个 γ，使得 $v \triangleright \text{模式} = \gamma$。

最后，我们现在可以通过步关系定义语义；步关系规定了执行当前可能的状态转移之一后，全局状态会发生什么变化。

定义 2.9（步关系）。给定协议 $\langle(S_i, o_i, F_i, \mathscr{L}_{!i}, \mathscr{L}_{?i}, \delta_i, <_i)\rangle_{i=1}^{N}$，关系[1] step 表示整个并发系统的非确定性转移，类型为 $\text{step} : G_N \to \mathscr{P}(G_N)$。如果 c_2 是 $\text{step}(c_1)$ 的可能结果之一，我们将 $\text{step}(c_1) \ni c_2$ 写作如下，由以下两条规则定义：

$$\frac{\delta_i(s_i, P!t) = s_i' \qquad \gamma_i(P) = j \qquad \gamma_i(t) \to^* v}{step(\langle(s_i, m_i, \gamma_i)\rangle_{i=1}^{N}) \quad \ni \quad \langle \ldots, (s_i', m_i, \gamma_i), \ldots, (s_j, m_j \cdot [v], \gamma_j), \ldots \rangle} \text{(SEND)}$$

$$\frac{\delta_j(s_j, ?pat) = s_j' \qquad m_j = m \cdot [v] \cdot m' \qquad pick(s_j, m, v) = \langle pat, \gamma \rangle}{step(\langle(s_i, m_i, \gamma_i)\rangle_{i=1}^{N}) \quad \ni \quad \langle \ldots, (s_j', m \cdot m', \gamma_j \cup \gamma), \ldots \rangle} \text{(RECV)}$$

[1]注意，步骤可以等价地视为一个非确定性函数（产生多个可能的全局状态），也可以视为从单个输入全局状态到多个结果全局状态的关系。此处的定义以声明式方式定义了这一关系。

第一条规则 (SEND) 对进程标识符环境 γ_i 中状态为 s_i 的进程 i 执行一个标签为 P!t 的发送转移$_i'$。我们从进程标识符环境中查找变量 Pγ_i，以获取要向其发送消息的进程标识符，即 γ_i(P) = j，这意味着我们向进程 j 发送消息。注意，由于 γ_i 是一个有限映射，若 P 不在 γ_i的定义域中，则 γ_i(P) =P。按照惯例，γ(t) 会将 t 中所有出现的 γ变量替换掉，从而允许使用内部状态；然后我们对所得项求值，得到一个值 v（用归约关系 $\rightarrow^*$表示）。随后，在结果全局配置中，进程 i 现在处于状态 s_i'，进程 j 的配置中则在其邮箱末尾加入了消息 v，而其他任何进程的状态都保持不变。

第二条规则 (RECV) 表示：进程 j 处于状态 s_j 时，在邮箱 m_j 中某处存在一条消息 v，其前缀为 m，后缀为 m′，则该进程执行标签为 pat 的接收转换，并进入状态 s_j'。我们使用 pick 函数检查语义约束：我们要选取第一条能够与某个可能模式匹配的消息 v，并且选取第一个满足条件的模式。如果给定状态、前缀和值后，这就是正确的模式，那么值项 v 将与 pat 合一，生成绑定 γ。随后，进程 j 进入状态 s_j'，消息 v 从其邮箱中移除，其进程标识符环境更新为 $\gamma_j \cup \gamma$：接收消息时匹配到的任何项都会更新其内部状态。除 j 以外的其他进程状态均保持不变。

接下来，描述计算可能结果的一种方式是使用轨迹，它表示从初始全局状态到某个可能最终状态的一系列步骤（由于每一步都具有非确定性，因此可能出现多种结果）。下面我们正式定义轨迹这一概念。

定义 2.10（轨迹）。轨迹是系统配置的一个序列，其中第一个是初始全局状态，后续状态则通过对前一配置应用步关系得到。如果全局状态序列 T 的形式为 T $= \langle t_0, t1, ..., t_x \rangle$，并且满足以下条件，我们就称 T 为轨迹：

- $t_0 = \langle (oi\ \varepsilon, /0) \rangle_{Ni=1}$
- 步骤(ti) $\ni$ti$_{+1}$
- step(t_x) $= /0$ where t_x 是轨迹序列中的最后一项。

例 2.2。回顾例 2.1 中的 mem 函数，我们为其分配进程标识符 0。我们考虑一个由四台机器组成的协议，其中另外三台机器的进程标识符分别为 1、2 和 3，定义如下：

$$S_1 = \{c_0, c_1, c_2, c_3\} \quad o_1 = c_0 \quad F_1 = \{c_3\} \quad \mathscr{L}_{!1} = \{\{\texttt{get, 1}\}, \{\texttt{put, X+1}\}\} \quad \mathscr{L}_{?1} = \{\texttt{X}\}$$
$$\delta_1(c_0, 0!\{\texttt{get, 1}\}) = c_1 \quad \delta_1(c_1, ?\texttt{X}) = c_2 \quad \delta_1(c_2, 0!\{\texttt{put, X+1}\}) = c_3$$

$$S_2 = \{d_0, d_1, d_2, d_3\} \quad o_2 = d_0 \quad F_2 = \{d_3\} \quad \mathscr{L}_{!2} = \{\{\texttt{get, 2}\}, \{\texttt{put, X+2}\}\} \quad \mathscr{L}_{?2} = \{\texttt{X}\}$$
$$\delta_2(d_0, 0!\{\texttt{get, 2}\}) = d_1 \quad \delta_2(d_1, ?\texttt{X}) = d_2 \quad \delta_2(d_2, 0!\{\texttt{put, X+2}\}) = d_3$$

$$S_3 = \{e_0, e_1\} \quad o_3 = e_0 \quad F_3 = \{e_1\} \quad \mathscr{L}_{!3} = \{\{\texttt{put, 0}\}\} \quad \mathscr{L}_{?3} = \emptyset$$
$$\delta_3(e_0, 0!\{\texttt{put, 0}\}) = e_1$$

上面的第一台机器（进程标识符为 1）向进程 0 发送“get”请求，随后接收结果 X，并向 0 发送一条带有 X+1 的“put”消息。上面的第二台机器（进程标识符为 2）与第一台类似，向进程 0 请求该值，然后向 0 发送一条带有 X+2 的“put”消息。第三台机器（进程标识符为 3）向 0 发送一条带有初始值 0 的“put”消息。

然后，我们可以得到协议 $\langle (S_i, o_i, F_i, \mathscr{L}_{!i}, \mathscr{L}_{?i}, \delta_i) \rangle_{i=1}^{3}$ 的以下轨迹；这只是众多可能性之一，展示了移动性以及存储内部状态的能力（为清楚起见，我们在轨迹的每一步中用下划线标出配置中发生变化的部分）：

$\langle(s_0,\varepsilon,\emptyset),$ $(c_0,\varepsilon,\emptyset),$ $(d_0,\varepsilon,\emptyset),$ $(e_0,\varepsilon,\emptyset)\rangle$
$\langle\underline{(s_0,[\{\texttt{put},0\}],\emptyset)},$ $(c_0,\varepsilon,\emptyset),$ $(d_0,\varepsilon,\emptyset),$ $\underline{(e_1,\varepsilon,\emptyset)}\rangle$
$\langle\underline{(s_0,\varepsilon,\{S\mapsto 0\})},$ $(c_0,\varepsilon,\emptyset),$ $(d_0,\varepsilon,\emptyset),$ $(e_1,\varepsilon,\emptyset)\rangle$
$\underline{\langle(s_0,[\{\texttt{get},1\}],\{S\mapsto 0\})},$ $\underline{(c_1,\varepsilon,\emptyset)},$ $(d_0,\varepsilon,\emptyset),$ $(e_1,\varepsilon,\emptyset)\rangle$
$\langle\underline{(s_1,\varepsilon,\{S\mapsto 0,P\mapsto 1\})},$ $(c_1,\varepsilon,\emptyset),$ $(d_0,\varepsilon,\emptyset),$ $(e_1,\varepsilon,\emptyset)\rangle$
$\langle\underline{(s_0,\varepsilon,\{S\mapsto 0,P\mapsto 1\})},$ $\underline{(c_1,[0],\emptyset)},$ $(d_0,\varepsilon,\emptyset),$ $(e_1,\varepsilon,\emptyset)\rangle$
$\langle(s_0,\varepsilon,\{S\mapsto 0,P\mapsto 1\}),$ $\underline{(c_2,\varepsilon,\{X\mapsto 0\})},$ $(d_0,\varepsilon,\emptyset),$ $(e_1,\varepsilon,\emptyset)\rangle$
$\langle(s_0,[\{\texttt{put},1\}],\{S\mapsto 0,P\mapsto 1\}),$ $\underline{(c_3,\varepsilon,\{X\mapsto 0\})},$ $(d_0,\varepsilon,\emptyset),$ $(e_1,\varepsilon,\emptyset)\rangle$
$\underline{\langle(s_0,[\{\texttt{put},1\},\{\texttt{get},2\}],\{S\mapsto 0,P\mapsto 1\})},$ $(c_3,\varepsilon,\{X\mapsto 0\}),$ $\underline{(d_1,\varepsilon,\emptyset)},$ $(e_1,\varepsilon,\emptyset)\rangle$
$\langle\underline{(s_0,[\{\texttt{get},2\}],\{S\mapsto 1,P\mapsto 1\})},$ $(c_3,\varepsilon,\{X\mapsto 0\}),$ $(d_1,\varepsilon,\emptyset),$ $(e_1,\varepsilon,\emptyset)\rangle$
$\langle\underline{(s_1,\varepsilon,\{S\mapsto 1,P\mapsto 2\})},$ $(c_3,\varepsilon,\{X\mapsto 0\}),$ $(d_1,\varepsilon,\emptyset),$ $(e_1,\varepsilon,\emptyset)\rangle$
$\langle\underline{(s_0,\varepsilon,\{S\mapsto 1,P\mapsto 2\})},$ $(c_3,\varepsilon,\{X\mapsto 0\}),$ $\underline{(d_1,[1],\emptyset)},$ $(e_1,\varepsilon,\emptyset)\rangle$
$\langle(s_0,\varepsilon,\{S\mapsto 1,P\mapsto 2\}),$ $(c_3,\varepsilon,\{X\mapsto 0\}),$ $\underline{(d_2,\varepsilon,\{X\mapsto 1\})},$ $(e_1,\varepsilon,\emptyset)\rangle$
$\langle\underline{(s_0,[\{\texttt{put},3\}],\{S\mapsto 1,P\mapsto 2\})},$ $(c_3,\varepsilon,\{X\mapsto 0\}),$ $\underline{(d_3,\varepsilon,\{X\mapsto 1\})},$ $(e_1,\varepsilon,\emptyset)\rangle$
$\langle\underline{(s_0,\varepsilon,\{S\mapsto 3,P\mapsto 2\})},$ $(c_3,\varepsilon,\{X\mapsto 0\}),$ $(d_3,\varepsilon,\{X\mapsto 1\}),$ $(e_1,\varepsilon,\emptyset)\rangle$

3 刻画 CAA 系统：无竞争与收敛

我们将探讨如何刻画 CAA 之间的竞争条件，并找出一类无竞争或具有收敛性的 CAA 系统。为刻画竞争条件，我们首先定义这样一个概念：在轨迹的特定步骤中，进程可能观察到哪些消息作为“传入”消息。我们通过消息多重集来定义这一概念：

定义 3.1（邮箱中的消息多重集）。给定一个邮箱 m，我们用 A_m 表示 m 中元素的多重集，即存在一种将 A_m 中的元素“排列”起来以得到 m 的方式。
定义 3.2（传入消息多重集）。设 $G_N = \langle c1, ..., (s_i, m_i, \gamma_i) ..., c_N\rangle$ 是一个全局配置，其中进程 i 处于状态 s_i，且系统的下一步会产生 M 个可能的配置：

$$step(G_N) = \{\langle c_{11},...,(s_{i1},m_{i1},\gamma_{i1}),...,c_{N1}\rangle,...\langle c_{1M},...,(s_{iM},m_{iM},\gamma_{iM}),...,c_{NM}\rangle\}$$

传入消息多重集 I_i 表示进程 i 的所有可能传入消息，定义如下：

$$I_i = (\bigcup_{y=1}^{M} A_{m_{iy}}) - A_{m_i}$$

也就是说，我们取进程 i 执行一步后所得到的所有可能邮箱多重集 $A^{m_{iy}}$ 的并集，再减去该步骤执行前 i 的邮箱中的消息（多重集差）。换言之，传入消息 I_i 是执行一步后可能添加到 i 的邮箱中的所有消息。

I 的定义隐式地以起始配置 G_N 为参数。通常，我们会用上标表示 I 在轨迹中某个位置 x 处出现，即 I_i^x。
备注。我们使用重载记法 $m_i = \langle I_i^0, I_i^1, ..., I_i^n\rangle$ 来表示一个邮箱，其中第一条消息取自 I_i^0，第二条取自 I_i^1，依此类推。也就是说，$m_i = \langle l_{0i}, l_{1i}, ..., l_{ni}\rangle$，其中 $l_{xi} \in I_i^x$。

进程i的邮箱现在可以写成传入消息的多重集序列，更准确地说，是 $\langle I_i^0$、I_i^1、...、$I_i^n\rangle$的一个子序列。因此，$m_i = \langle I^{x_0}{}_i \ , I^{x_1}{}_i \ \cdots, I^{x_{m_i}}{}_i \ \rangle$，其中 $x_0 < ... < x_m$。但为什么是子序列？如果消息 l_{xi} 被消费，那么多重集 I_i^x就会从邮箱中消失。

注（转移函数δ表示法重载）。通常，转移函数 δ 的参数是一个状态和一个标签组成的对。我们重载该对的第二部分，使其允许为一个多重集，使得$\delta(s,?I_i^x) = \{s' \mid \forall l \in I_i^x, \delta(s,?l) = s'\}$，即 I_i^x 中任意可能消息的目标状态集合。

定义 3.3（竞争条件）。对于一个协议，竞争条件表示这样一种情形：存在某个状态，它可以从同一个 I_i^x 消息邮箱集合中消耗 2 条或更多消息，并且无法消耗先前邮箱集合中的任何消息。也就是说：

$$\forall y \in [0, x-1].\ |\delta(s,?I_i^y)| = 0 \wedge |\delta(s,?I_i^x)| \geq 2$$

直观地说，如果在进程的当前状态下，我们可以从集合 I_i^x 接收 2 条或更多消息，就会面临竞争条件，因为这些消息是在同一步发送的，且可能以任意顺序到达。这由上述合取式的第二部分表示。不过，为了到达 I_i^x，我们需要先不消耗任何消息，因此有合取式的第一部分。如果先前的消息多重集只有一条可供我们消耗的消息，就不会发生竞争条件。

双自动机系统的轨迹收敛与无竞争性 我们考虑一类无竞争的二元 CAA 系统（即 $N = 2$），并通过证明其轨迹总是收敛来说明这一点，也就是说，该系统是确定性的。此外，这使得双自动机系统的测试容易得多，因为我们只需检查一条轨迹，就能确定系统的最终状态。

我们回顾一下，CAA 的定义仅允许一组受限的转移：如果某个状态存在转移，那么这些转移只能是按正确顺序排列的若干个接收转移，或者一个单独的发送转移。事实上，Erlang 的语义规定，转移不能以其他方式混合，并且从任何状态出发至多只能有一个发送转移，我们将这一条件称为仿射发送。我们利用这一条件来推理轨迹中出现非确定性的可能性。

以下我们假设不允许自我消息传递：即对于任意局部状态 (s_i, m_i, γ_i)，若 $s_i \in S_i$, $\delta(s_i, P!t)$ 对任意满足 $\gamma_i(P) = i$[2] 的 P 都未定义。对于不允许自我消息传递的二元模型类别，我们观察到（并将在下文正式证明）不会出现竞争条件。竞争条件（定义 3.3）的一个前提是，参与者能够在同一轨迹步骤接收两条传入消息。根据上述条件，同一时刻至多只有一个消息发送方，因此，在所有全局配置中，参与者的每个状态至多对应一个转移。因此，如果到达某个最终状态，那么到达该状态的可能轨迹就只有一条。

为了证明参与者在特定时刻至多有一个转移，我们需要考察邮箱。发送动作是仿射的（即确定性的），因此我们只需证明接收转移是确定性的。证明考虑系统存在两个不同全局配置的两种可能情况：

- 两个配置中的自动机都处于即将发送的状态，或都处于即将接收的状态。由于两个自动机将执行相同类型的动作，它们只会影响一个邮箱；在这种情况下，邮箱配置不会发生分歧。

- 一个自动机处于即将发送的状态，而另一个处于接收状态。如果处于接收状态的参与者没有可供消费的有效消息，那么唯一有效的动作就是发送（即另一个配置取得进展）；否则，两个动作都会影响同一个邮箱，从而使这两个配置发生分歧。然而，发送状态是一种“建设性”动作，它会将消息追加到邮箱末尾；而接收动作则是“破坏性”动作，会从邮箱中消费一条消息，它们彼此不相交的性质最终将导致邮箱收敛。

[2]此限制旨在避免向自身发送静态消息（例如 1!t）和动态消息（例如 P!t，其中 P 绑定到 1）这两种情况。更严格的做法是要求每个进程只向彼此发送静态消息。

命题 3.1（收敛）。设 $X_1 = (S_1, o_1, F_1, \mathscr{L}_{!_1}, \mathscr{L}_{?_1}, \delta_1)$ 和 $X_2 = (S_2, o_2, F_2, \mathscr{L}_{!_2}, \mathscr{L}_{?_2}, \delta_2)$ 构成一个由两个参与者组成的协议，其中不存在混合转换，只有仿射发送，且没有自我消息传递。如果此协议收敛到最终状态，则该过程是确定性的，即任何全局配置都会收敛到同一个最终全局配置，因此该协议是无竞争的。

证明。我们通过证明任意两个大小相同的轨迹 T_1 和 T_2 相同，来证明所需结论。注意，可能的状态序列必须遵循模式 $\langle s0, ..., s_n \rangle$，其中 n 为某个数；我们证明，对于任意 $k \leq n$，T_1 和 T_2 的长度为 k 的前缀相同。取 k 为 n，即可得出二者相同。此证明通过对 k（其有上界）进行良基归纳来完成：

1. 基本情况：我们取 $k = 1$。那么两个前缀都应为 $\langle s \rangle$，其中 s 是初始全局状态。这由轨迹的定义可知，因为对于任意给定配置，它确定性地规定了第一个状态。 2. 归纳步骤：我们的归纳假设表明，我们的前缀 $\langle s_0 ... s_k \rangle$ 匹配。如果 $k = n$，那么我们就完成了。然而，如果 $k \leq n$，那么 s_k 不是最终状态，并且我们有 $T_1 = \langle s_0 ... s_k t_1 \rangle$和 $T_2 = \langle s_0 ... s_k t_2 \rangle$。现在我们必须证明 $t_1 = t_2$。由于最多只能有另一个进程发送消息，因此我们有 $\forall j \in [1, k+1], |I_{j+1} i| \leq 1$。现在，由于集合的大小至多为一，消耗一条消息会使该集合为空，因此我们可以将邮箱表示为集合 $\{I_i^0, I_i^1, .., I_i^k\}$ 的子序列，其中 $m_i = \langle I^{x_1} i \ , I^{x_2} i \ \cdots I^{x_y} i \ \rangle$，其中 $x_1 < x_2 < ... < x_y < k+1$。我们找出最大的 z，使得 $\forall j \in [1, z], |\delta(s_k, ?I_j i)| = 0$。令 P_i 为接收转换之后所有可能状态的集合。如果 $j = y$，那么 $|P_i| = 0$，因此无法消耗消息，我们停留在同一状态；否则 $P_i = \{s' | \forall m \in I_i^{z+1}, \delta(s_k, ?m) = s'\}$。由于最多只能接收一条消息，$|I_i^{z+1}| \leq 1$ 因此 $|P_i| \leq 1$，至多有一个可能的转移，所以只有一个可能的新状态。这不可能为零，因为我们有 t_1 和 t_2：因此，可能的下一个状态列表只有一个成员，并且 $t_1 = t_2$，正如预期。

□

Erlang中的兼容性 关于通信有限状态机的早期研究指出，机器之间的兼容性概念是确保系统进展的关键步骤 [14]。兼容性是通常所称对偶性的自动机对应概念：每个接收都有相应的发送，反之亦然。一对兼容的确定性机器因而不会发生死锁和未指定接收 [14]。

由于 Erlang的设计原则，我们可能并不总是关心最终得到一个空邮箱，并且即使某些消息未被接收、仍有消息留在邮箱中，也承认系统具有兼容性。因此，对于未来的工作，我们提出系统终止条件的“层级”，大致刻画不同程度的兼容性。

下文中，令 CAA 系统的所有可能轨迹集合为 Y。

定义3.4。（第1级）系统是强兼容的，当且仅当，$\forall T \in Y, t_{|T|} = \langle (f_i, \varepsilon, \gamma_i) \rangle_{i=1}^{N}$ 其中 $f_i \in F_i$，即，没有进程的邮箱中还留有任何消息，且它们已达到终态。

定义3.5。（第2级）系统是弱兼容的，当且仅当，$\forall T \in Y, t_{|T|} = \langle (f_i, m_i, \gamma_i) \rangle_{i=1}^{N}$ 其中 $f_i \in F_i$，即某些进程的邮箱中可能还留有一些消息，但所有进程都已达到终态。

定义3.6。（第3级）当 $\forall T \in Y, t_{|T|} = \langle (s_i, \varepsilon, \gamma_i) \rangle_{i=1}^{N}$ where $s_i \in S_i / F_i$ 时，系统是通信受限的，即，进程尚未到达其终态，但由于缺少输入而无法继续。

定义 3.7.（第4级）如果不满足此前任何条件，则称该系统不兼容。

我们指出，例 2.2 中的系统是强兼容的，因为所有可能的轨迹最终都会以空邮箱和终态结束。

4 讨论与相关工作

Fowler 描述了一种框架，可根据 Scribble 语言 [12]中构想的多方会话类型，为 Erlang/OTP 的 gen_server 行为生成运行时监控器。该框架借鉴了 Deniélou 和 Yoshida [10]提出的思路：将 Scribble 的全局类型（多方会话类型）投影为局部类型，再将局部类型实现为通信有限状态机。不过，目前尚不清楚这种方法在多大程度上能够建模 Erlang 的通用邮箱语义。这值得进一步研究。

Erlang的一句经典箴言是“让它崩溃”（或“让它失败”）。我们这里的模型不处理进程失败，尽管近期的模型已纳入了这类方面 [4]。在 Bocchi 等人提出的模型中， [4], 参与者通过单向链接与其邮箱进行通信，这与通信有限状态机的结构相似，但在步骤发生的方式上具有更大的灵活性。该方法未整合模式匹配或动态拓扑结构。邮箱 MSC（消息序列图） [5, 6], 构建了一个进程的消息序列图模型，该模型具有单个传入通道，其匹配语义在理念上与我们的模型相似，但并不直接基于通信有限状态机传统。与我们方法进行更深入的比较是未来的工作。我们方法的一个显著区别在于，通过为每个进程设置“记忆”环境 γ来整合动态拓扑结构，这使得消息可以发送给一个变量，而该变量是由先前的接收操作绑定的进程标识符。

在初步工作中，我们创建了一个工具，可以从 Erlang 代码中提取 CAA 模型，也可以反过来，根据 CAA 协议的描述生成并发通信代码的 Erlang 骨架。后续工作包括将其开发为用于分析和规约 Erlang程序的工具。我们还注意到，由于可以描述移动进程，我们推测 Milner 将 λ-演算翻译为 π-演算的 CPS 翻译或许可以调整为以 CAA 为目标语言。如果确实如此，那么 CAA 所描述的就是一种图灵完备的计算模型，我们计划进一步研究这种可能性。

致谢 感谢 Kartik Jalal 分享从 Erlang 中提取 CAA 模型的见解，感谢 Simon Thompson 在这一构想初期提供 Erlang 专业知识，也感谢 2017年至2020年间选修 CO545：函数式与并发编程课程的肯特大学本科生，作为该模型及其在诊断竞争条件和死锁方面的试用者。此项工作部分由 EPSRC 项目 EP/T013516/1（Granule）资助。Orchard 还部分获得 Eric 和 Wendy Schmidt 的慷慨支持，该支持由 Schmidt Futures 项目推荐提供。

参考文献

[1] Joe Armstrong（1997）：Erlang 的发展。载于：第二届 ACM SIGPLAN 函数式编程国际会议论文集，第 196–203 页，doi:10.1145/258948.258967。[2] Joe Armstrong（2007）：Erlang 的历史。载于：第三届 ACM SIGPLAN 编程语言史会议论文集，第 6–1 页，doi:10.1145/1238844.1238850。[3] Samik Basu、Tevfik Bultan 与 Meriem Ouederni（2012）：判定编排的可实现性。ACM SIGPLAN 通知 47(1)，第 191–202 页，doi:10.1145/2103621.2103680。[4] Laura Bocchi、Julien Lange、Simon Thompson 与 A. Laura Voinea（2022）：参与者与灰色故障模型。载于 Maurice H. ter Beek 与 Marjan Sirjani 编：协调模型与语言——第24届 IFIP WG 6.1 国际协调会议（COORDINATION 2022），作为第17届国际分布式计算技术联合会议（DisCoTec 2022）的一部分举行，意大利卢卡，2022年6月13–17日，论文集，《计算机科学讲义》13271，施普林格，第 140–158 页，doi:10.1007/978-3-031-08143-9_9。

[5] Benedikt Bollig、Cinzia Di Giusto、Alain Finkel、Laetitia Laversa、Etienne Lozes 与 Amrita Suresh（2021）：判定可同步性的统一框架。载于：CONCUR 2021——第32届并发理论国际会议，第 1–33 页，doi:10.4230/LIPIcs.CONCUR.2021.14。[6] Ahmed Bouajjani、Constantin Enea、Kailiang Ji 与 Shaz Qadeer（2018）：关于有界异步性下验证消息传递程序的完备性。载于：计算机辅助验证：第30届国际会议（CAV 2018），作为 FLoC 2018 联合逻辑会议的一部分举行，英国牛津，2018年7月14–17日，论文集，第二部分 30，施普林格，第 372–391 页，doi:10.1007/978-3-319-96142-2_23。[7] Daniel Brand 与 Pitro Zafiropulo（1983）：关于通信有限状态机。《ACM学报》（JACM）30(2)，第 323–342 页，doi:10.1145/322374.322380。[8] Richard Carlsson（2001）：Core Erlang 入门。载于：PLI 论文集，1，Citeseer。[9] Richard Carlsson、Björn Gustavsson、Erik Johansson、Thomas Lindgren、Sven-Olof Nyström、Mikael Pettersson 与 Robert Virding（2000）：Core Erlang 1.0语言规范。乌普萨拉大学信息技术系，技术报告。[10] Pierre-Malo Deniélou 与 Nobuko Yoshida（2012）：多方会话类型与通信自动机相遇。载于：欧洲编程研讨会，施普林格，第 194–213 页，doi:10.1007/978-3-642-28869-2_10。[11] Alain Finkel 与 Etienne Lozes（2017）：通信有限状态机的可同步性不可判定。arXiv 预印本 arXiv:1702.07213，doi:10.48550/arXiv.1702.07213。[12] Simon Fowler（2016）：多方会话参与者的 Erlang 实现。arXiv 预印本 arXiv:1608.03321，doi:10.4204/EPTCS.223.3。[13] Mohamed G Gouda（1984）：闭覆盖：验证通信有限状态机的进展。IEEE 软件工程汇刊 (6)，第 846–855 页，doi:10.1109/TSE.1984.5010313。[14] Mohamed G Gouda、Eric G Manning 与 Yao-Tin Yu（1984）：关于两个有限状态机之间通信的进展。《信息与控制》63(3)，第 200–216 页，doi:10.1016/S0019-9958(84)80014-5。[15] Julien Lange、Emilio Tuosto 与 Nobuko Yoshida（2015）：从通信机器到图形化编排。载于 Sriram K. Rajamani 与 David Walker 编：《ACM SIGPLAN-SIGACT编程语言原理研讨会》第42届年会论文集（POPL 2015），印度孟买，2015年1月15–17日，ACM，第 221–232 页，doi:10.1145/2676726.2676964。[16] Jan Pachl（2003）：通信有限状态机的可达性问题。arXiv 预印本 cs/0306121，doi:10.48550/arXiv.cs/0306121。[17] Wuxu Peng 与 S Purushothaman（1992）：一类通信有限状态机的分析。《信息学学报》29(6-7)，第 499–522 页，doi:10.1007/BF01185558。[18] Louis E Rosier 与 Mohamed G Gouda（1984）：判定一类通信有限状态机的进展。