- **semantics**
  - **operational**
  - **denotational : what we’re doing here?**
  - **axiomatic : Hoare?**
- **process algebra**

Cut

- **Based on Alur's 1999**
- **A subset of events E is a cut c iff:**
  forall e1 and e2 in E, if e1 in c and e2 < e1, then e2 in c

- **Problem: given a cut c and an event e, is cU{e} a cut?**
  - **Brute force: power set? no**
  - **Property of partial order relation: with appropriate data structure, O(1)**
    - **There are only two kinds of events: sends and receives**
    - **By transitivity, for sends, we need only to work out the immediate ancestor, and the rest follows by induction. For receives, additionally, check if the cut contains the corresponding send**
  - **We can easily get the immediate ancestor of an event by keeping track of the last event on that instance; < need not be completely known**

MSC Syntax & Semantics

ITU

- **Z.120**
  - **BNF syntax**
  - **Graphical and concrete syntax**
  - **informal static requirements**
  - **examples**
- **Annex B**
  - **defined on modified subset of Z.120 syntax**
  - **informal semantics**
  - **process algebraic formal semantics**
  - **NO formal static requirement**

Alternative Formalizations

Process Algebra <- recommendation standard

- An Algebraic Semantics of Basic Message Sequence Charts
- High-level Message Sequence Charts
- Operational Semantics for MSC'96
- Z.120 Annex B: Formal semantics of Message Sequence Charts

Petri Nets

- Petri nets as a Semantic Model for Message Sequence Chart Specifications
- Translating Message Sequence Charts to other Process Languages using Process Mining

Buchi Automata

- Interpreting message flow graphs
- Modeling, specifying, and verifying message passing systems

Partial Order

- An Analyzer for Message Sequence Charts
- (Alur's 1999) Model Checking of Message Sequence Charts *(+ automata) <-
- (Alur's 2003) Implied message sequence charts *(+ concurrent automata) <- our choice
- A theory of regular MSC languages
- Automata and Logics for Timed Message Sequence Charts

MSC Formalism

Alur's 2003

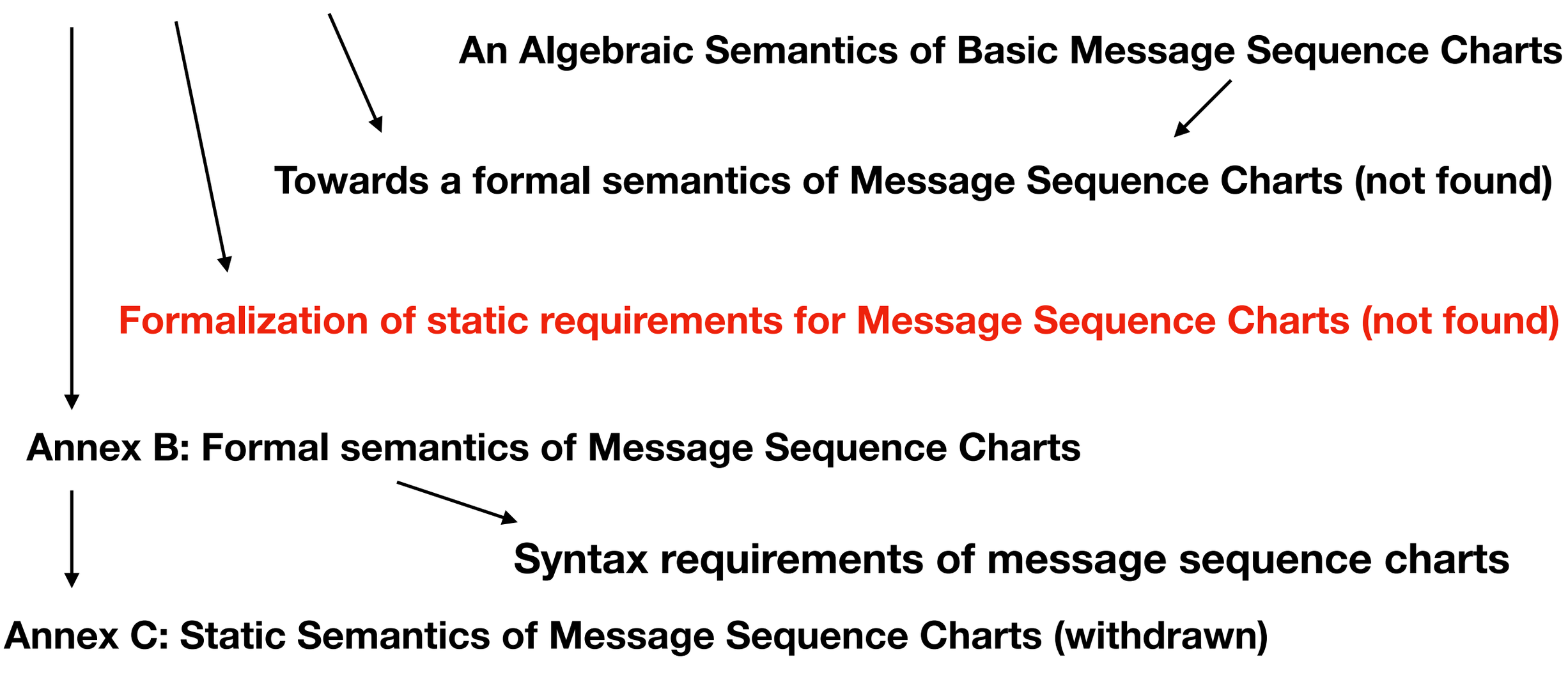
1. **Based on MSC formalism in Alur's 1999**
2. **Σ consists of send(i,j,a) / receive(i,j,a)**
3. **Non-degeneracy: no message overtaking**
4. **Linearization / Word: linear extension**
5. **Count: #(w, x) = number of x in w**
6. **Projection: w|i = events in w of process i ; M|i = events in M of process i**
7. **Well-formed: all receives have sends ; Possible: event safe to add**
8. **Complete: all sends have receives**
9. **MSC: <M|i | i of [n]> or <w|i | i of [n]> uniquely determines a MSC ([n]=process num)**
10. **Interleaving Closure:**
    1. **w and v are equivalent if they determine the same MSC**
    2. **<w> = set of all words that determine the same MSC**
    3. **L satisfies closure if forall w of L, <w> is a subset of L**
11. **L is a MSC-language iff:**
    1. **L contains only well-formed and complete words**
    2. **L satisfies closure**
12. **Partial MSC:**
    1. **Determined by a well-formed word v**
    2. **MSC M is a completion of v if all of v's projections is prefixes of M's**

Question: Is this good enough? Are the semantic requirements sufficient? Automata?

Static Requirements

1. **Current state of affairs: no official formal semantic constraints defined by ITU**
2. **Tracing old papers: there WERE formal semantic constraints but none found**
3. **Important people: Sjouke Mauw and Michel Reniers**
4. **Touched on by and partially provided in**
    1. **The formalization of Message Sequence Charts**
    2. **Syntax requirements of message sequence charts**

The formalization of Message Sequence Charts



Attempt at Formalization

Here's the plan:

1. **Use Alur's as base and extend it to include the additional stuff in Z.120**
2. **Comply with the informal static requirements in Z.120 and Annex B**
3. **Put it together with model learning**

Standard doesn't care to formalize static requirement, so there's no need?

Attempt at Formalization: Things Missing

Basic MSC

- **Environment**
- **Action**
- **Creation / Termination**
- **Timer**
- **Control Flow**
- **Incomplete message**
- **Condition**
- **General ordering**

Structural constructs

- **Coregion**
- **Inline expression: no idea**
- **MSC reference**
- **Instance Decomposition**
- **HMSC**
- **Horizontal composition / Alternative composition**

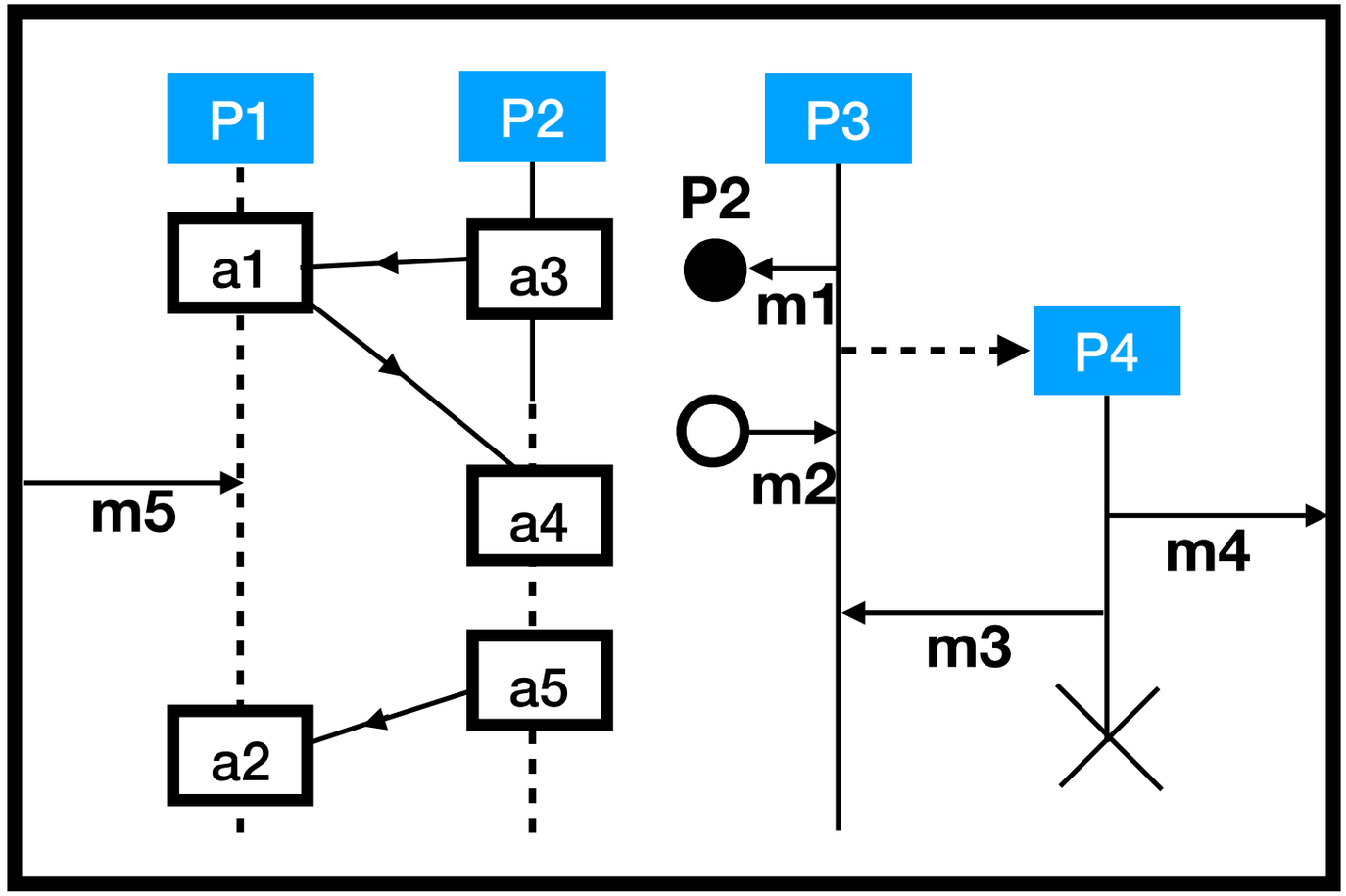
Other stuff

- **Gate**
- **Data**
- **Time**
- **MSC Chart Document**

Attempt at Formalization: Semantics of Things Missing

Basic MSC semantics

- **Environment : a special instance without local order that doesn't own events**
- **Action : events local to a instance**
- **Creation / Termination : dynamically created instance by others or self-terminate**
- **Timer : start / stop / timeout (what does it do?)**
- **Control Flow : blocking/non-blocking method call (later?)**
- **Incomplete message : sent can be lost and received can be found**
- **Condition : set or guard (somewhat formalized? more on this later)**
- **General ordering : explicit ordering between events that are otherwise unordered**



Attempt at Formalization: Semantics of Things Missing (c.)

Structural constructs semantics

- **Coregion : part of instance that's not ordered**
- **Inline expression : high-level stuff; eg concat, loop, etc. (save for HMSC?)**
- **MSC reference : MSC in a another MSC (save for HMSC?)**
- **Instance Decomposition : i don't know (?)**
- **HMSC : like in Alur's 1999**

Other semantics

- **Gate : interface to other MSC; unnecessary to formalize?**
- **Data : interface for externally defined data language; how about just label event?**
- **Time : a lot of concepts for quantified time like special time events**
- **MSC Chart Document : meta info about MSC like instance kinds**

Question: do we need process algebraic semantics?

Won't consider MSC-graph or HMSC yet

Attempt at Formalization: Extended Formalization

A MSC is:

- **a process set P, two distinct processes env and unknown**
- **an event set E = S U R U S' U R' U A U CRE U TER ; they're disjoints ; |S| = |R|**
- **an event ownership function O : E -> P**
- **a process ownership function EoP : P -> E set**
- **a label set Σ**
- **a labeling function L : E -> Σ**
- **a bijective function f : S -> R**
- **a general ordering (strict partial order) relation <G**
- **for each p ∈ P, a local coregion set of event sets Cp**
- **for each p ∈ P, a visual local (strict partial order) relation <p (disregard coregion)**
- **a visual global (strict partial order) relation <**

- **< = transitive closure of ((Up<p) U {(s, f(s)) | s ∈ S} U <G U**
  {(c,e) | c ∈ CRE, e ∈ EoP(O(c))} U {(e,t) | t ∈ TER, e ∈ EoP(O(t))} -
  (Up{(e1,e2) | e1≠e2, (e1,e2) ∈ cxc, c ∈ Cp}))

Additional Constraints:

- **coregion closure: (coregion must be contiguous)**
  for all p, for all c ∈ Cp, for all (e1,e2) ∈ cxc, if e1≠e2, for all e in EoP(p),
  if e1≠e and e2≠e, (e1,e) ∈ <p and (e,e2) ∈ <p implies e ∈ c
- **if coregion closure holds and < is not a strict partial order, <G is illegal**

Attempt at Formalization: Extended Formalization (cont.)

Elements in the label set Σ each matches the form of exactly one of these:

- **send(i, j, m), i≠j and (i,j) ∈ PU{env}xPU{env} - (env,env) and m ∈ E**
- **receive(i, j, m), i≠j and (i,j) ∈ PU{env}xPU{env} - (env,env) and m ∈ E**
- **send'(i, j, m), i≠j and (i,j) ∈ PxPU{env,unknown} - (env,env) and m ∈ E**
- **receive'(i, j, m), i≠j and (i,j) ∈ PU{env,unknown}xP - (env,env) and m ∈ E**
- **action(i, a), i ∈ P and a ∈ E**
- **create(i, j), i≠j and (i,j) ∈ PxP**
- **terminate(j), j ∈ P**

L and O respect the semantic correspondence:

- **if e ∈ S, L(e) = send(i, j, m), O(e) = i**
- **if e ∈ R, L(e) = receive(i, j, m), O(e) = j**
- **if e ∈ S', L(e) = send'(i, j, m), O(e) = i**
- **if e ∈ R', L(e) = receive'(i, j, m), O(e) = j**
- **if e ∈ A, L(e) = action(i, a), O(e) = i**
- **if e ∈ CRE, L(e) = create(i, j), O(e) = i**
- **if e ∈ TER, L(e) = terminate(j), O(e) = j**

Additional Constraints:

- **if e ∈ CRE and L(e) = create(i, j), it's the unique e such that L(e) = create(i, j)**
- **if e ∈ TER and L(e) = terminate(j), it's the unique e such that L(e) = terminate(j)**

Attempt at Formalization: Checking Static Requirements

Message

- **For send(i, j, m) and receive(i, j, m), i and j are either env or process sat by Σ**
- **send-receive causal dependency consistency sat by $\{(s, f(s)) \mid s \in S\} \subseteq <$**
- **send-receive unique mapping sat by f plus $|S| = |R|$**
- **For send'(i, j, m), i is either env or process and j is either env, unspecified, process**
  For receive'(i, j, m), it's the same with i and j switched places sat by Σ

Action

- **For action(i, a), i is process sat by Σ**

General Ordering

- **Orders defined must be irreflexive. No loop or upward ordering on a process is allowed sat by making < strict**

Process Creation and Termination

- **For create(i, j), i must be process and j is the new process sat by Σ**
- **For terminate(j), j must be process sat by Σ**
- **For any process, it can only be created once sat by additional constraints**
- **No event before creation sat by <**
- **create(i, j) is associated with i sat by O**
- **For any process, it can only be terminated once sat by additional constraints**
- **No event after termination sat by <**
- **terminate(j) is associated with j sat by O**

Attempt at Formalization: w/ Model Learning

Without MSCs => mark undesirable state/output
Result: found bugs in TCP and TLS take multiple steps to manifest; eg SMACK
Note: not all illegal traces lead to undesirable stuff

With MSCs as spec => immediately find illegal transition by:

- **the same cut definition**
- **for Mealy, must first consider where exactly is SUL**
  - **find the instance or instances that's SUL**
  - **isolate them and treat the rest of the MSC as env, which will be the learner**
  - **a transition is illegal if it doesn't error for unsafe input or return unsafe output for safe input**
  - **redefine Mealy language: a string is (input.output)* ; a safe string is when all prefixes are cuts**

Question: is this too strict? like prune away too much information?

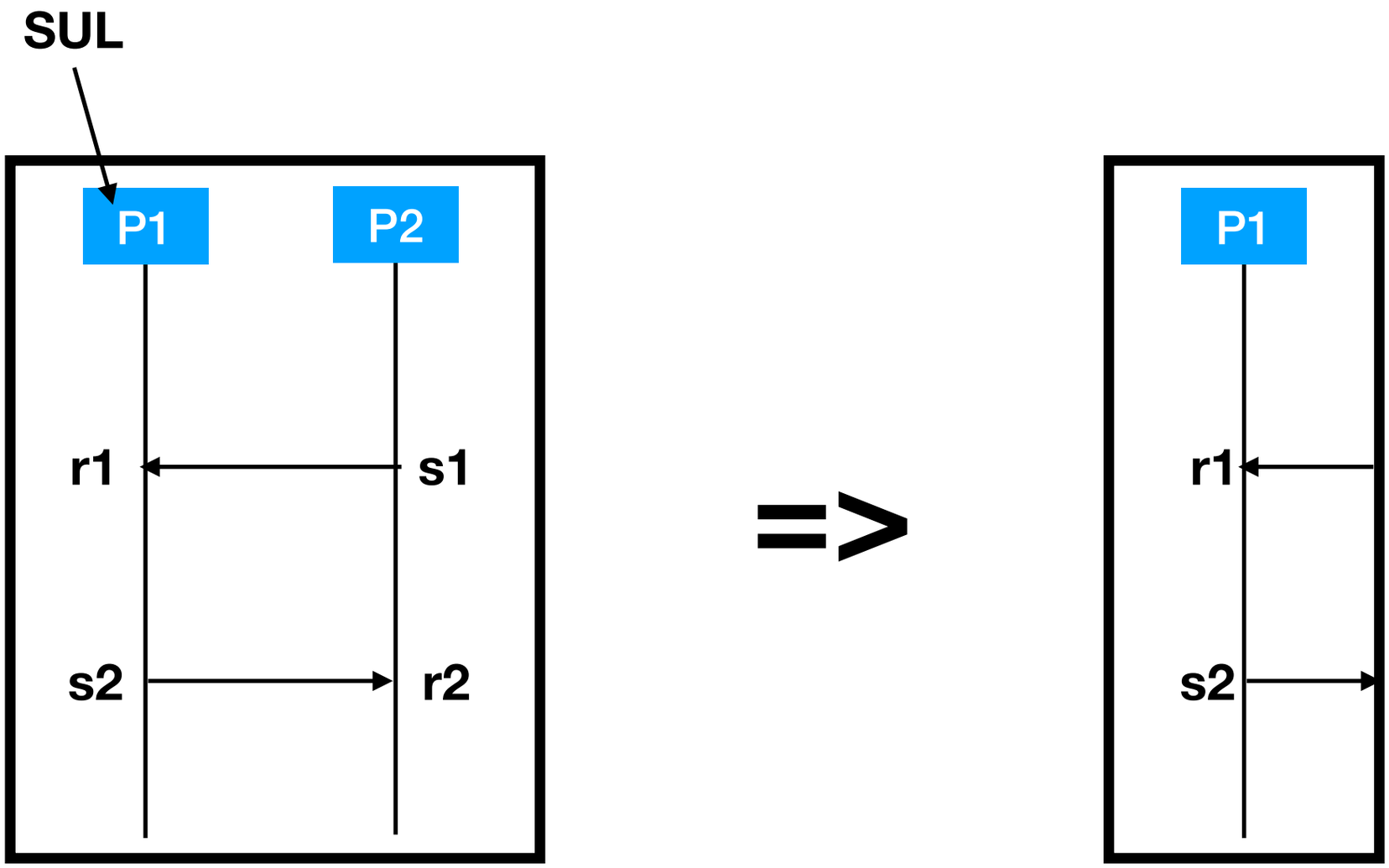
Problem: opaqueness of black box makes it impossible to know about events that don't concern env, such as actions or process creations/terminations

for following examples i consider empty(no response), failure, alert, etc. as okay error
also a problem: no way of telling empty and internal error

Attempt at Formalization: w/ Model Learning (cont.)

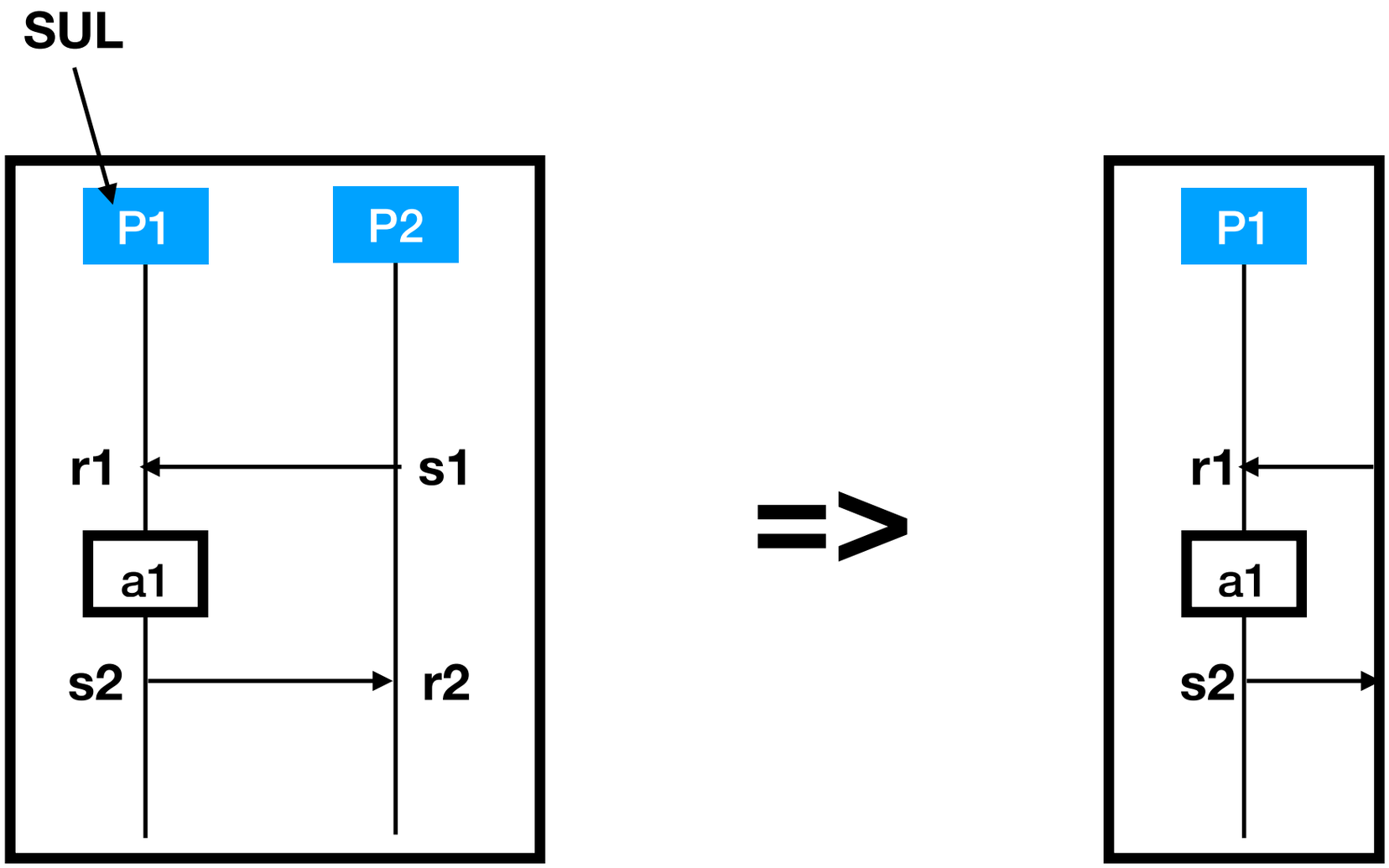
when there are only sends and receives:

- **inputs are always receives; outputs are always sends**
- **since learning connection is lossless, when leaner/mapper sends s1, we record r1 as input and when we receive receive r2 we record s2 as output**
- **e.g.: r1/s2 and r2/empty are safe ; r1/s1 and r2/s2 are not**

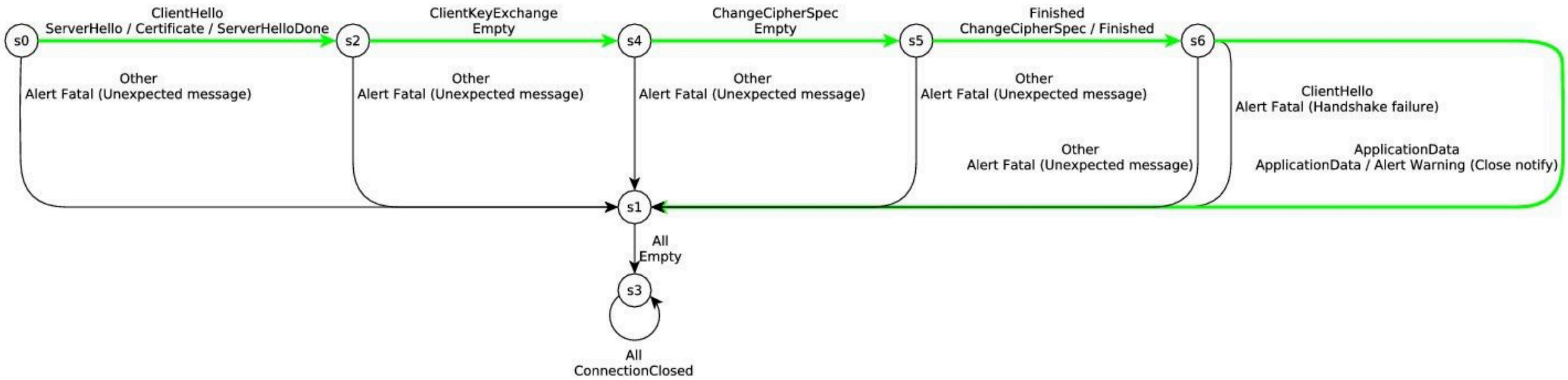


Attempt at Formalization: w/ Model Learning (cont.)

no way of telling if and when a1 takes place
e.g.: r1/s2 and r1/a1,s2 look the same to learner



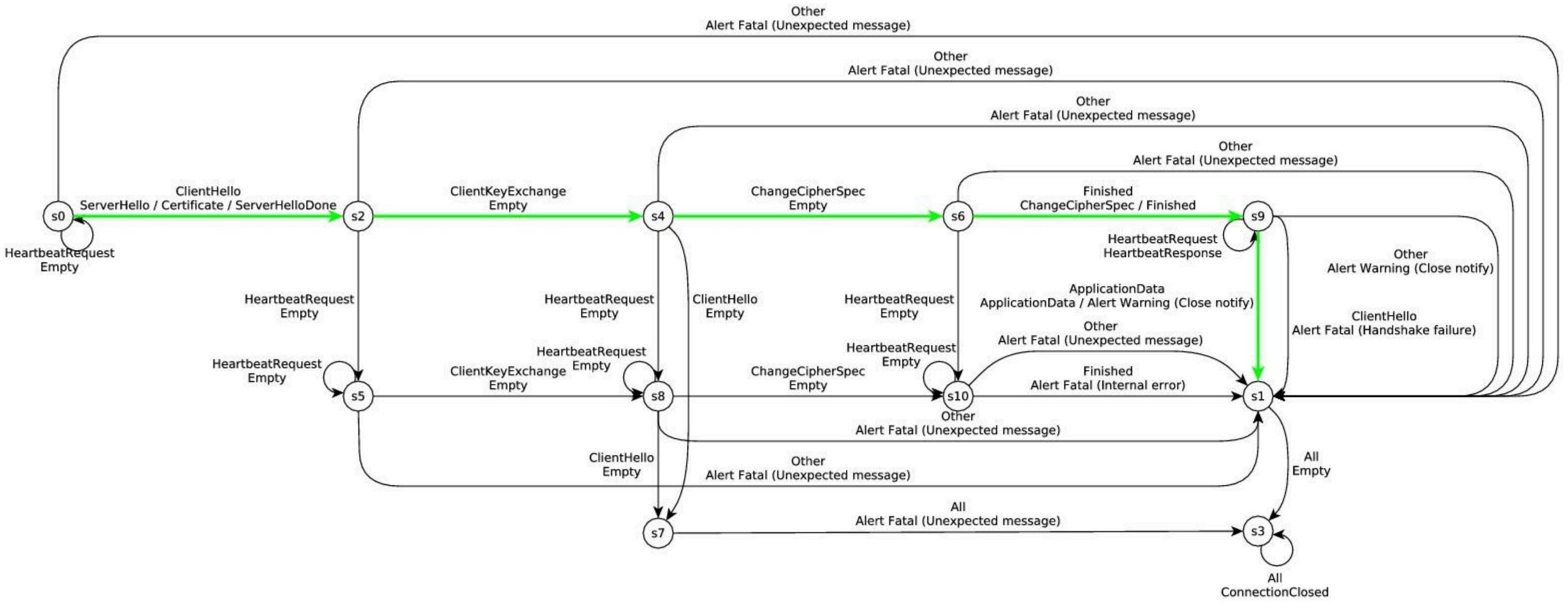
Learned NSS State Machine



it checks out

Learned GNU State Machine

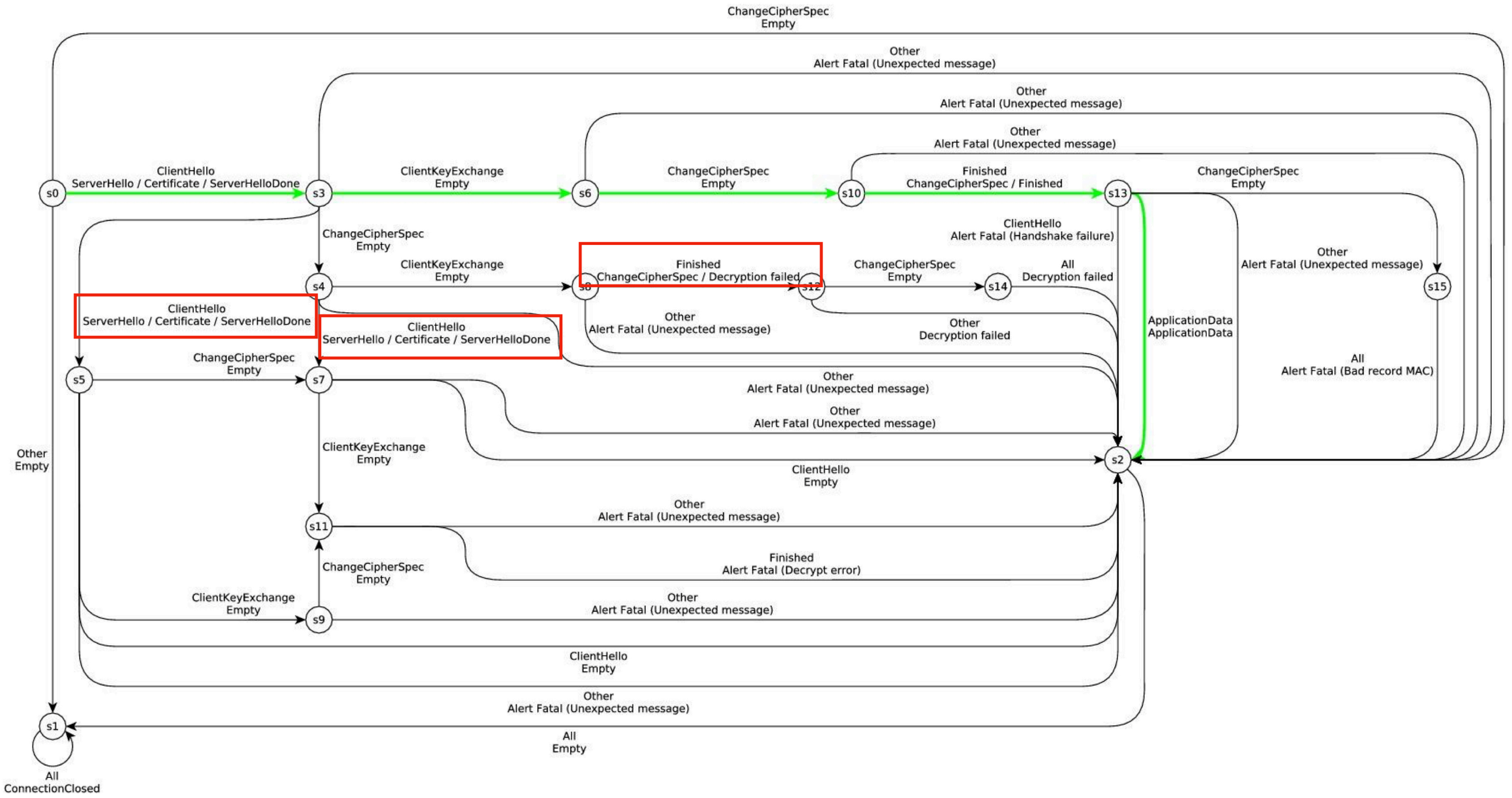
***using figs from slides not paper**



checks out other than heartbeat

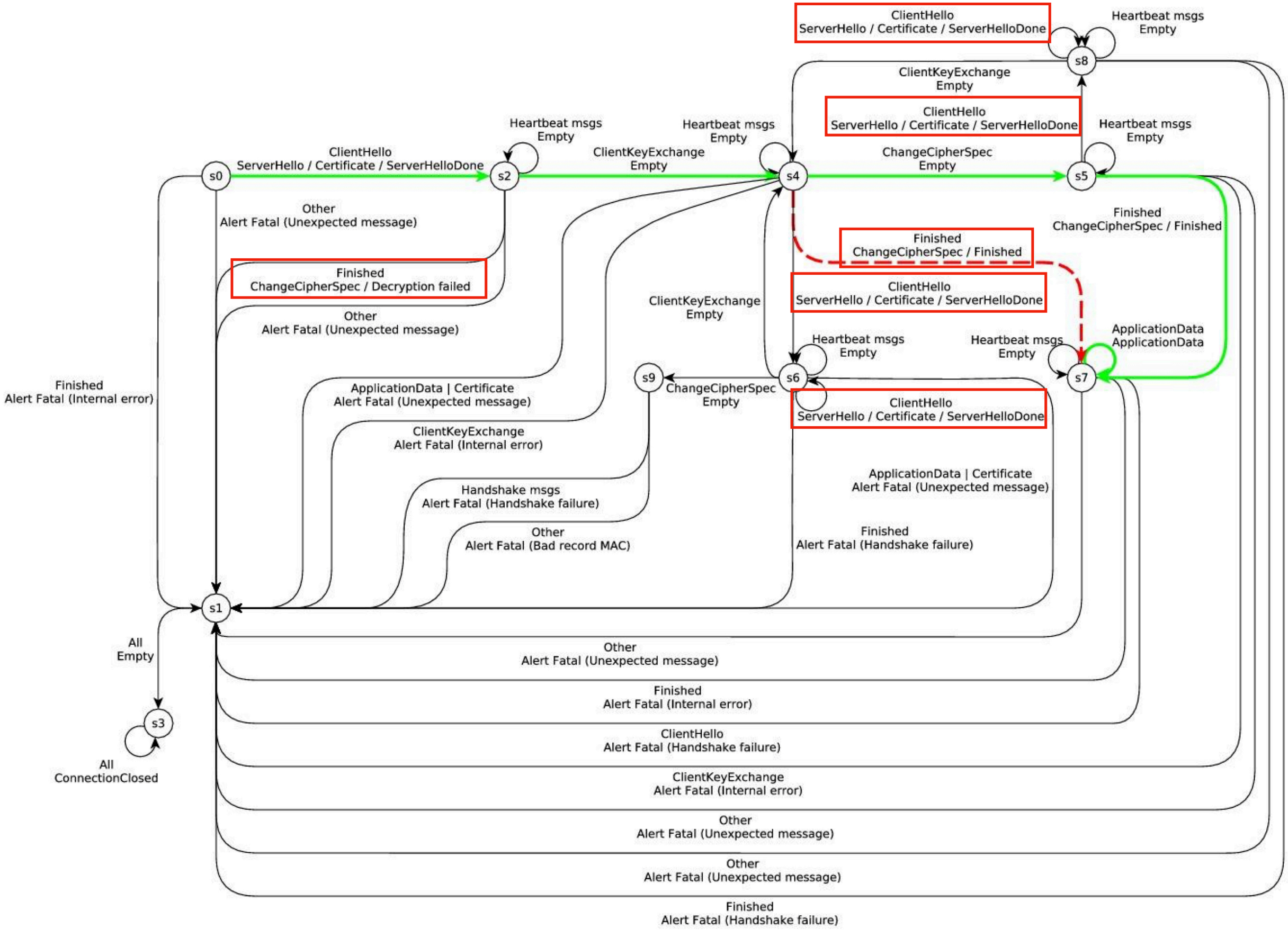
Learned OpenSSL State Machine

***using figs from slides not paper**



some weird response

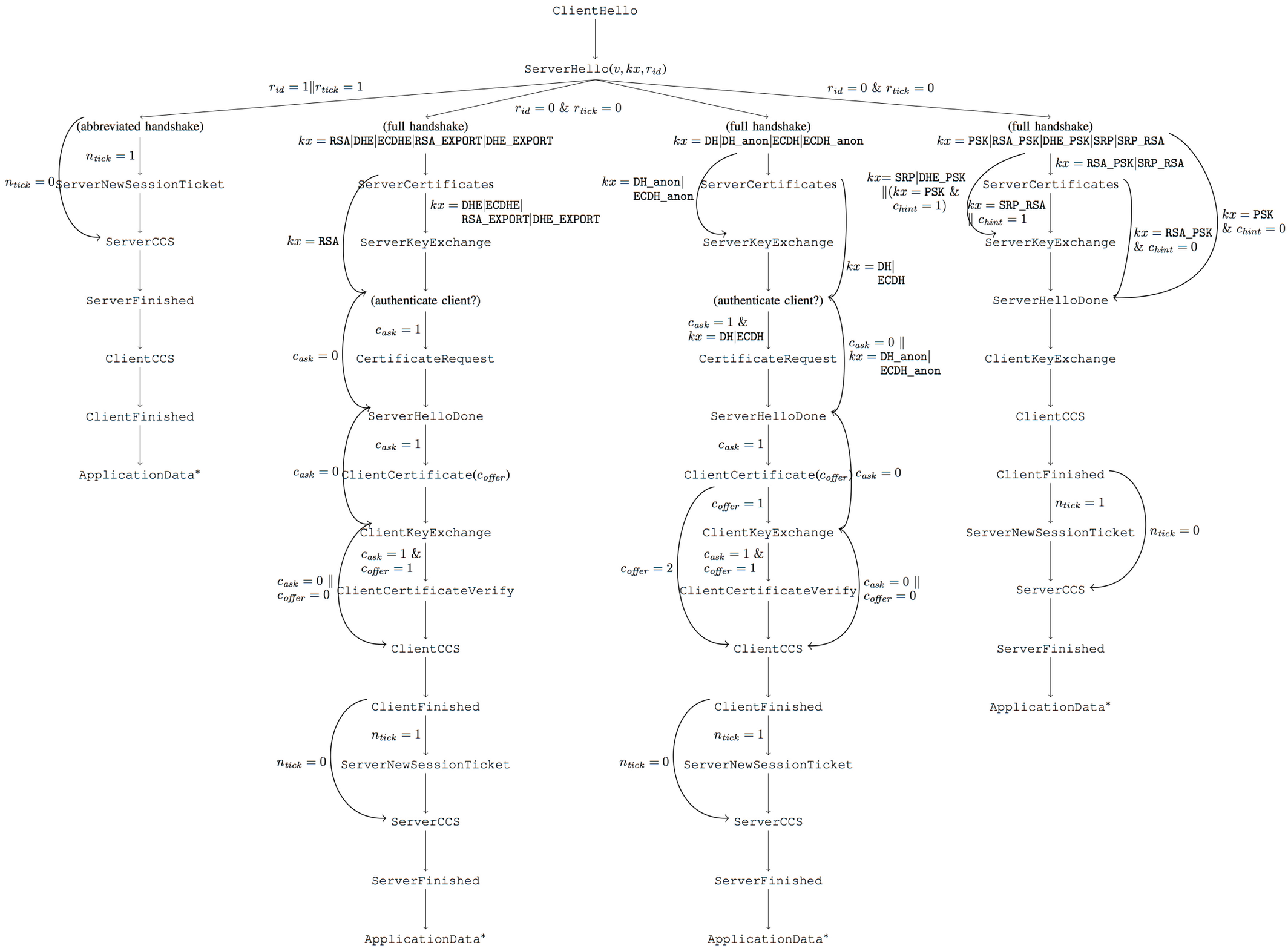
Learned JSSE State Machine



do we actually need all those MSC constructs?

what about conditions? taken care of by mapper?

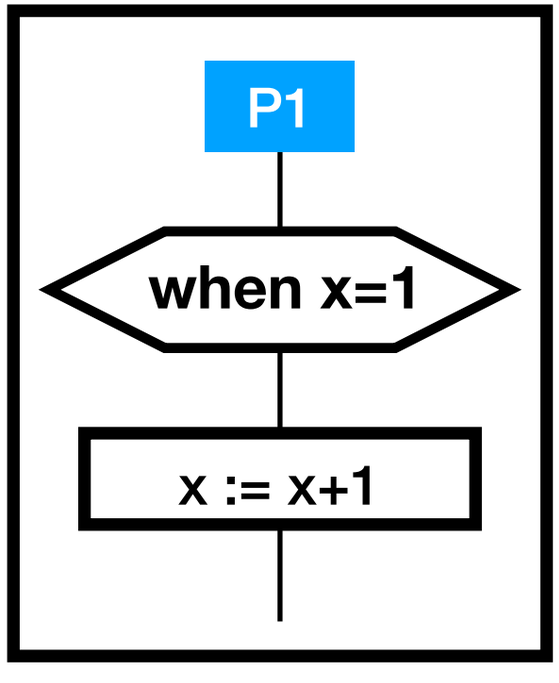
Handcrafted OpenSSL State Machine

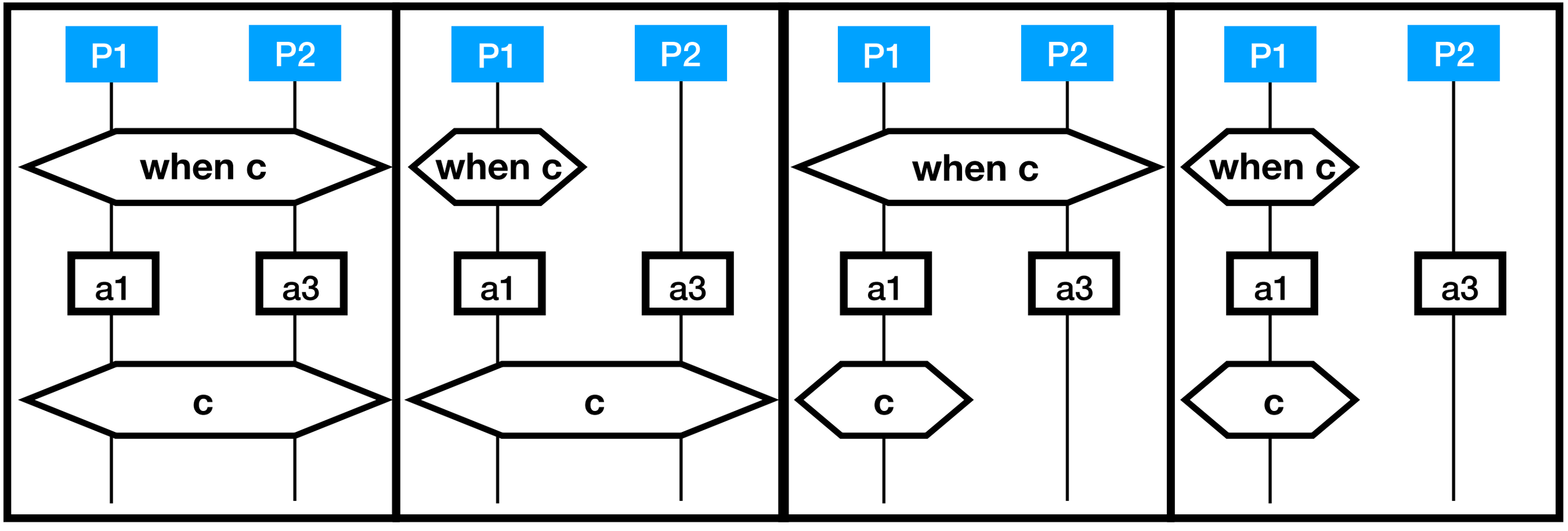


Condition

Problem

- **multiple instances?**
- **persistent memory**
- **data check?**
- **doesn’t make sense without higher-level or data**





Condition: Naive Approach

This is pretty stupid

- **for each p in powerset of P, maps it to a register set and a guard set**
- **don’t influence the ordering**

When doing model learning

- **when starting in a new MSC, check the guards**

should probably consider this in HMSC?

Condition: Word Approach

Word of MSC

- **in Alur's 2003, a MSC and a well-formed & complete word imply each other**
- **here, restrict a word to be only implied by a MSC, disregard the converse**
- **thus, a word w = w1w2...wn over E is just a linear extension of <**
- **well-formedness is ensured by having {(s, f(s)) | s ∈ S} ⊆ <**
- **completeness is ensured by definition of E where |S| = |R|**
- **interleaving closure of <w> is just L(M) for a MSC M**
- **a prefix word pw is a prefix of a word in L(M) ; PF(M) = set of all prefix words**
- **a suffix function; given a prefix word pw, SF(pw) = {sw | pw.sw ∈ L(M)}**

To add conditions to a MSC M and get M':

- **a set of conditions COND**
- **a set of condition setting events SET**
- **a set of condition guarding events GUARD**
- **M' is represented by a set of all its possible traces L(M'), which is found by:**
  - **put SET and GUARD to E and modify < as if they are plain actions and get M'**
  - **L(M') is just the union of two things:**
    - **{pw.SF(pw) | pw ∈ PF(M), pw[-1] ∈ GUARD, ∃c ∈ pw[:-1], c ∈ SET}**
    - **{w | w ∈ L(M), g ∈ GUARD, g ∉ w}**

(not sure if this works)

THE END