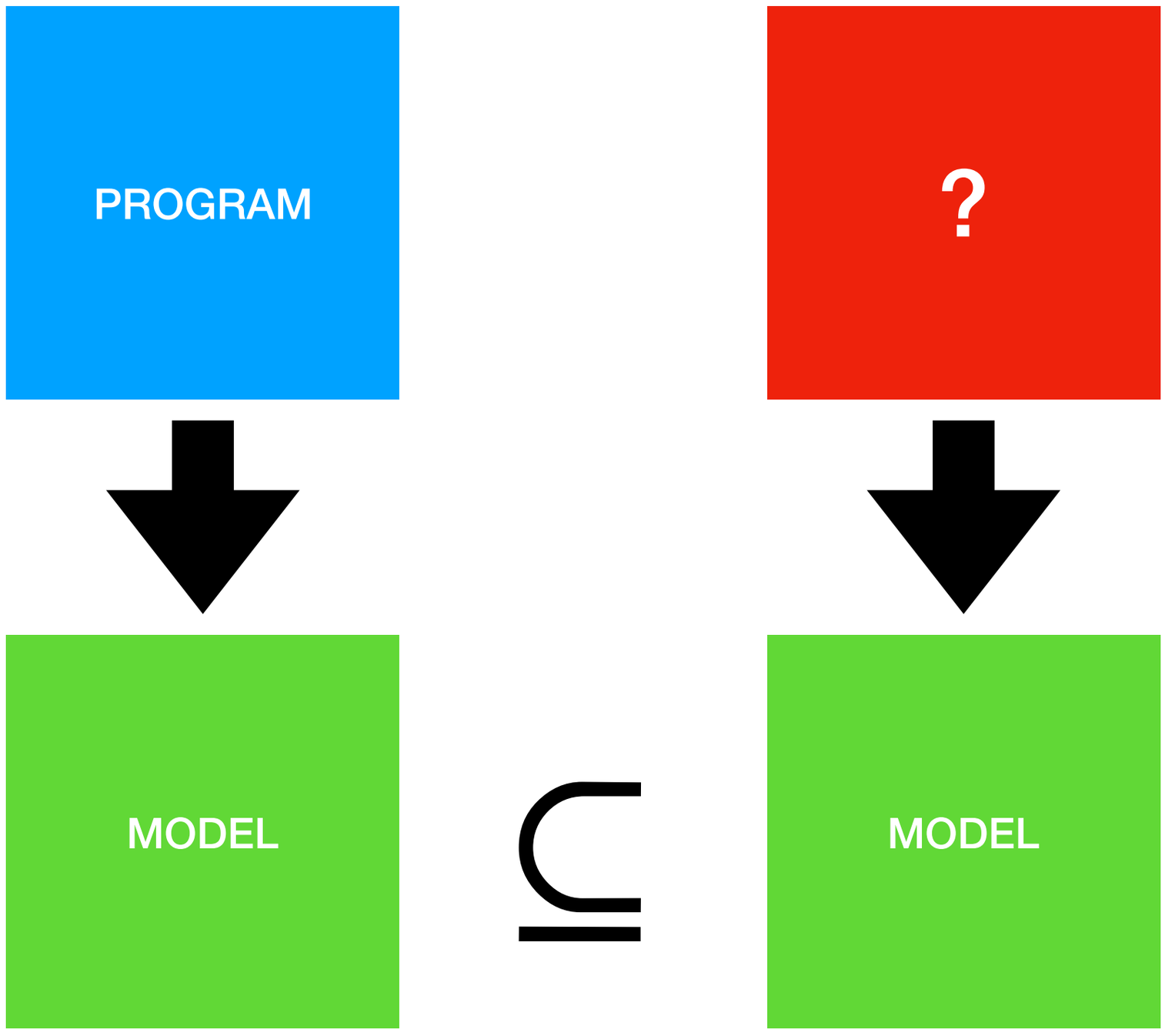
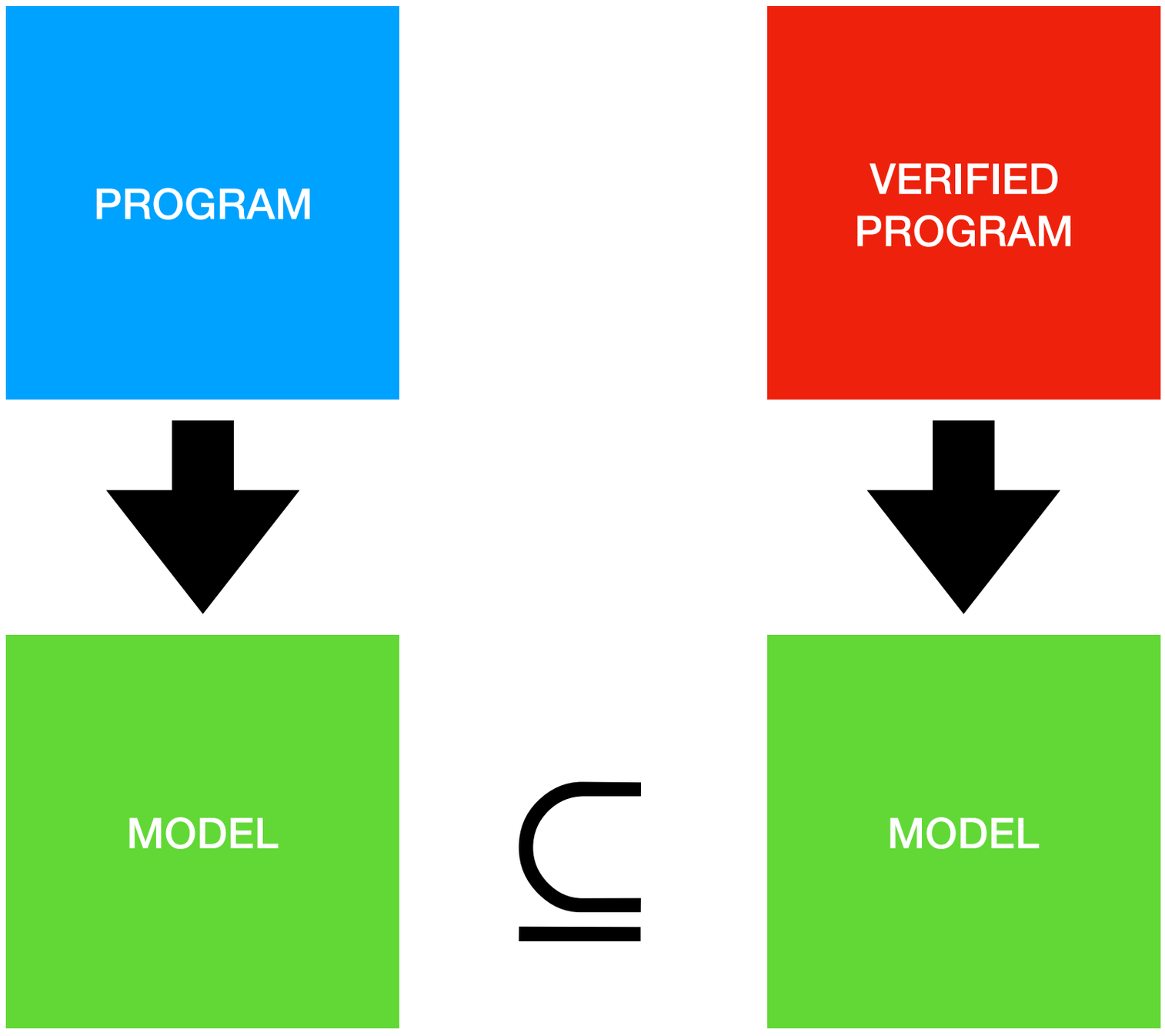
Overview



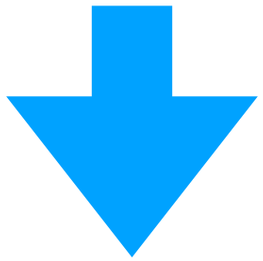
Digression: What if?



1. Unfinished business
    1. Angluin with Mealy
    2. Message synchronization
    3. Subject of learned model
2. MSC
    1. Formalism
    2. Model extraction
    3. Implementation problem
    4. Attempt at TCP
3. MSC-graph and HMSC
    1. Formalism
    2. Model extraction
    3. Attempt at TLS

Nerode Relation in Mealy Machines

- **Consider a DFA**
- **Exists a right invariant equivalence relation, RL, that's of finite index**
- **xRLy iff for all z of Σ*, xz in L <-> yz in L**



- **Consider a Mealy machine, with λ : I -> O**
- **Exists a right invariant equivalence relation, RL, that's of finite index**
- **xRLy iff for all z of I*, λ(xz) = λ(yz)**

- **Given some string, asking whether or not it's accepted by a DFA is analogous to asking what the output string is if passed to a Mealy machine.**
- **For a DFA, a transition taking it to a final state can be mapped to an output symbol T while a transition taking it to a non-final state can be mapped to an output F.**
- **For a Mealy machine, the input strings in its language take it to its final states, regardless of its outputs.**

equivalence: reflexivity, symmetry, transitivity / index: number of equivalence classes

Angluin: DFA v. Mealy

	Membership	Equivalence
DFA	Does SUL accepts a string s?	Are the languages of hypothesis and SUL exactly the same?
Mealy	What is the output sequence given a input sequence s?	For all input sequences, do hypothesis and SUL produce exactly the same output sequences?

Where Angluin uses {0,1}, Vaandrager uses O,
e.g., row function in Angluin's is (SUSA) -> E -> {0,1}
while in Vaandrager it's (SUSI) -> E -> O

1. **Unfinished business**
    1. Angluin with Mealy
    2. **Message synchronization**
    3. Subject of learned model
2. MSC
    1. Formalism
    2. Model extraction
    3. Implementation problem
    4. Attempt at TCP
3. MSC-graph and HMSC
    1. Formalism
    2. Model extraction
    3. Attempt at TLS

Timing of Messages

- **Problem 1: arbitrary timeout**
    1. **In "Protocol State Fuzzing of TLS Implementations", they introduce a distinct output symbol, Empty, which "is returned if no data is received from the SUT before a timeout occurs in the test harness". Its generation is managed by the test harness: "As the SUT will not always send a response [...] the test harness will generate a timeout after a fixed period. [...] As the timeout has a significant impact on the total running time we varied this per implementation."**
    2. **In "Combining Model Learning and Model Checking to Analyze TCP Implementations", they also introduce a special output symbol, timeout, produced by network adapter, "in case no packet was received within a predefined time interval." They admit this design may "sometimes missed packets generated by the SUL, reporting erroneous timeout outputs."**

- **Problem 2: sending another message before getting a response**
    - **In "Protocol State Fuzzing of TLS Implementations", it simply won’t happen because "After sending a message, the test harness waits to receive responses from the SUT."**
    - **In "Combining Model Learning and Model Checking to Analyze TCP Implementations", this concern is not mentioned.**

1. **Unfinished business**
    1. Angluin with Mealy
    2. Message synchronization
    3. **Subject of learned model**
2. MSC
    1. Formalism
    2. Model extraction
    3. Implementation problem
    4. Attempt at TCP
3. MSC-graph and HMSC
    1. Formalism
    2. Model extraction
    3. Attempt at TLS

TLS: Which side am I learning? Both, as a whole

- **From "Protocol State Fuzzing of TLS Implementations":**
  - **"When testing a server, the test harness is initialised by sending a ClientHello message to the SUT to retrieve the server's public key and preferred ciphersuite. When a reset command is received we set the internal variables to these values."**
  - **"To test client implementations we need to launch a client for every test sequence. This is done automatically by the test harness upon receiving the reset command. The test harness then waits to receive the ClientHello message, after which the client is ready to receive a query. Because the first ClientHello is received before any query is issued, this message does not appear explicitly in the learned models."**
  - **"In our analyses we use different input alphabets depending on whether we test a client or server, and whether we perform a more limited or more extensive analysis. To test servers we support the following messages: ClientHello [...] To test clients we support the following messages: ServerHello [...]"**

- **Conclusion: the learned model is one model that accounts for both client and server behaviors, but it is learned by splitting the alphabet into client side and server side, with the test harness corresponding to input symbols and SUL to output symbols, e.g., when harness is client and SUL is server, input consists of client-to-server messages and output consists of server-to-client messages.**
- **Confer "Lessons learned in the analysis of the EMV and TLS security protocols" for examples of learned models.**

TCP: Which side am I learning? Both, separately

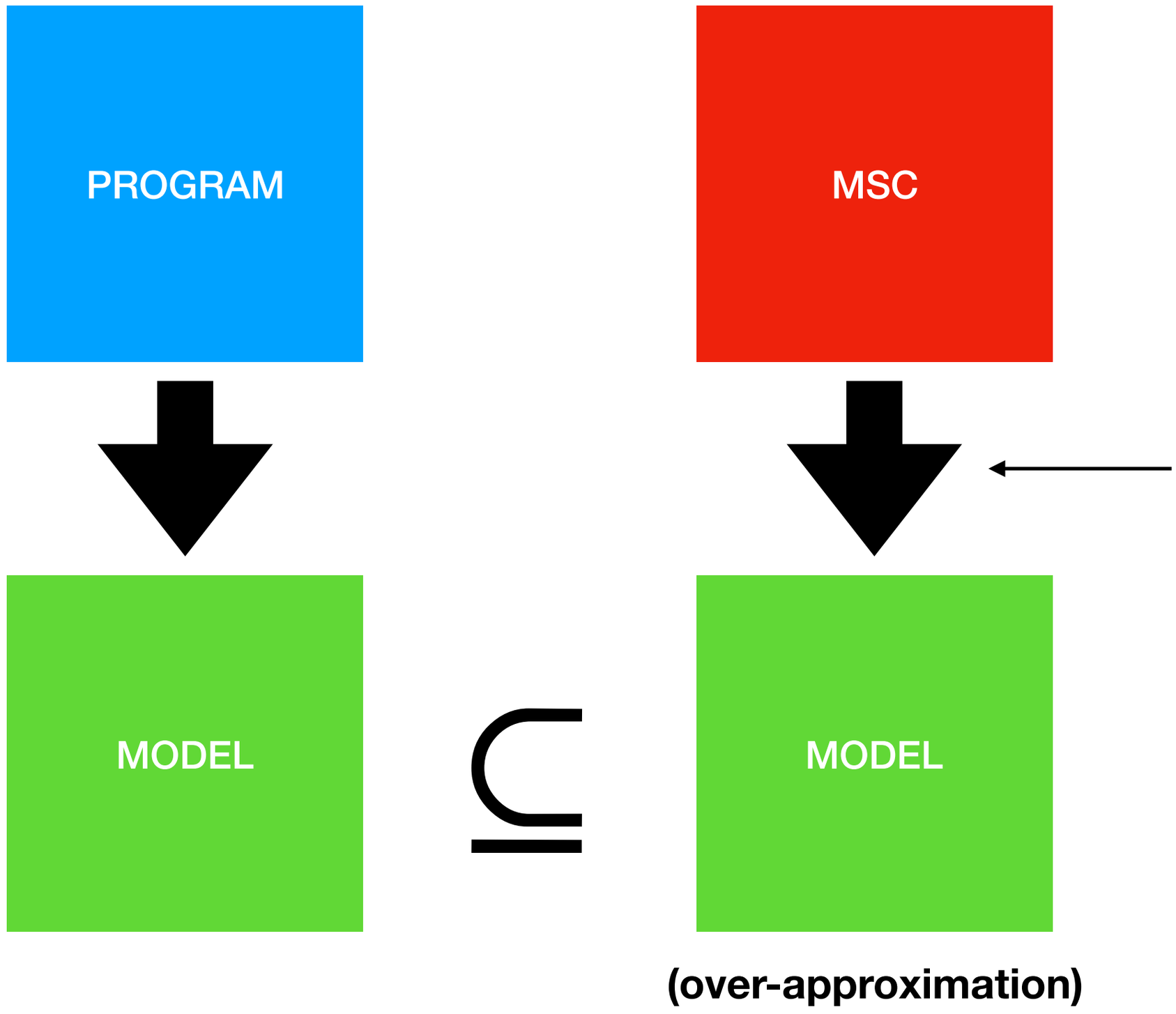
- **From "Learning Fragments of the TCP Network Protocol":**
  - **"[...] the SUT is the server in the TCP communication. On the other side, the learner and mapper simulate the client."**
  - **"[...] The server passively listens for incoming connections on a port while the learner, acting as a ”fake client”, sends messages to the server through its own port."**

- **From "Combining Model Learning and Model Checking to Analyze TCP Implementations"**
  - **"Our results considerably extend our previous work [...] since we have [...] inferred models of TCP clients in addition to servers [...]"**
  - **(On combining learned models of different implementations) "We composed pairs of learned client and server models with a hand-made model of a non-lossy network, which simply delivers output from one entity as input for the other entity."**

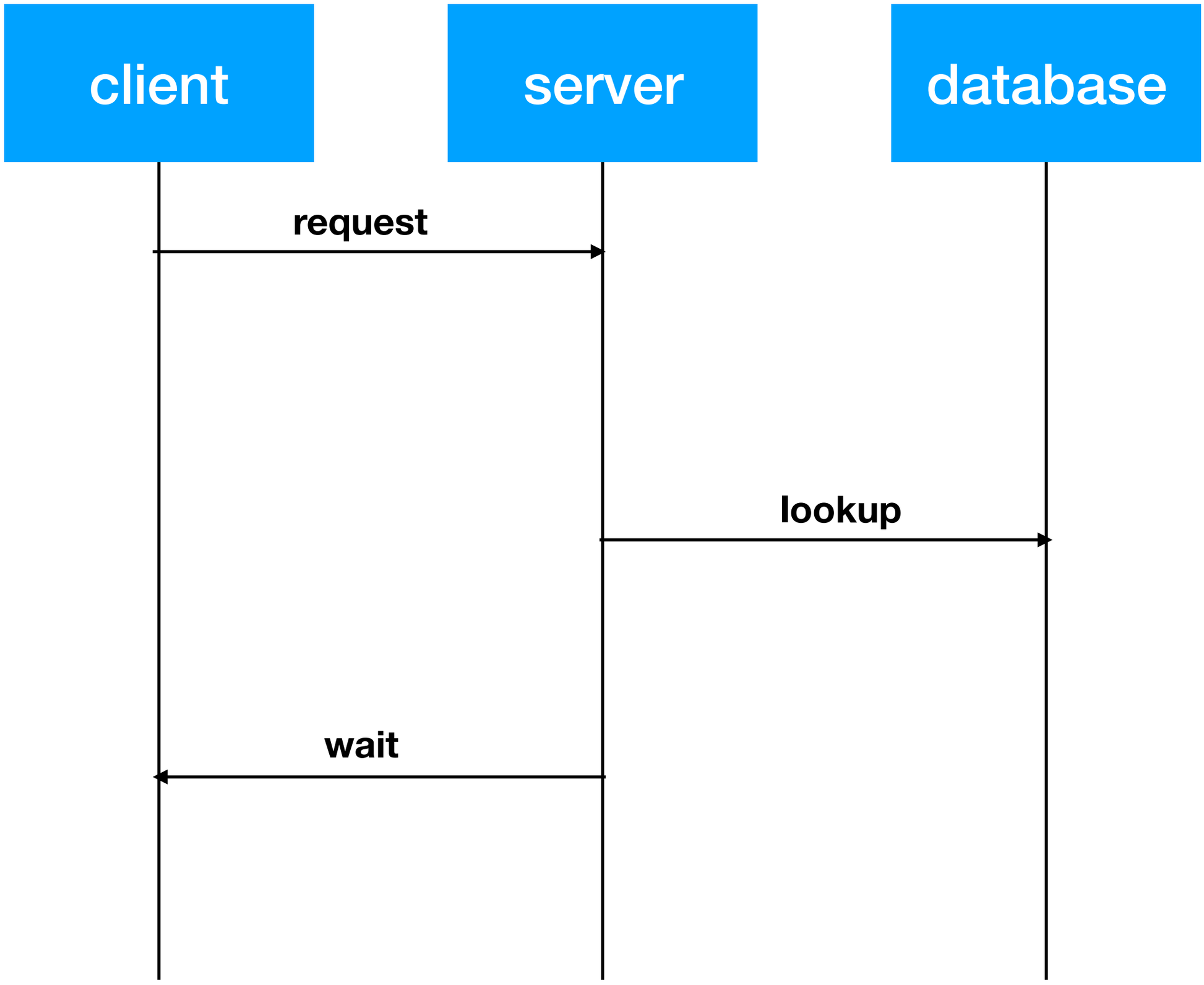
- **Conclusion: because whether it’s client or server matters, they are learned separately. When SUL is a server, the learned model takes client message as input and outputs server messages. This design allows for combining learned models from different server and client implementations.**

1. Unfinished business
    1. Angluin with Mealy
    2. Message synchronization
    3. Subject of learned model
2. **MSC**
    1. Formalism
    2. Model extraction
    3. Implementation problem
    4. Attempt at TCP
3. MSC-graph and HMSC
    1. Formalism
    2. Model extraction
    3. Attempt at TLS

Goal



MSC

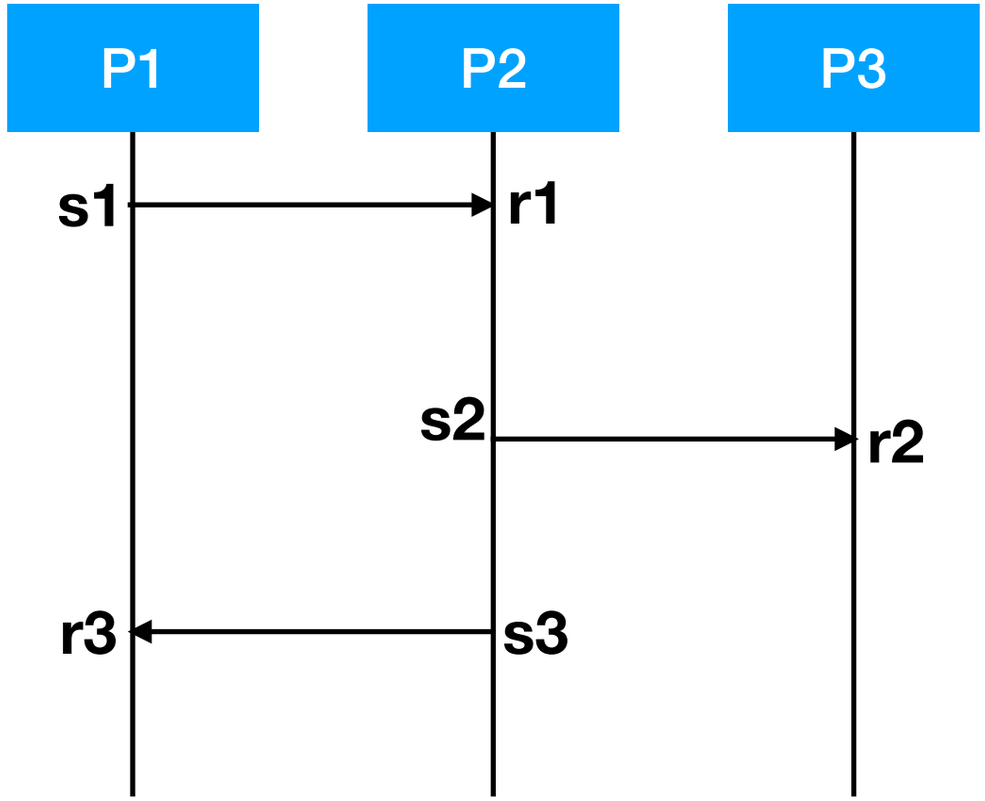


1. Unfinished business
    1. Angluin with Mealy
    2. Message synchronization
    3. Subject of learned model
2. MSC
    1. Formalism
    2. Model extraction
    3. Implementation problem
    4. Attempt at TCP
3. MSC-graph and HMSC
    1. Formalism
    2. Model extraction
    3. Attempt at TLS

Formalism (An Analyzer for Message Sequence Charts)

A MSC M consists of:

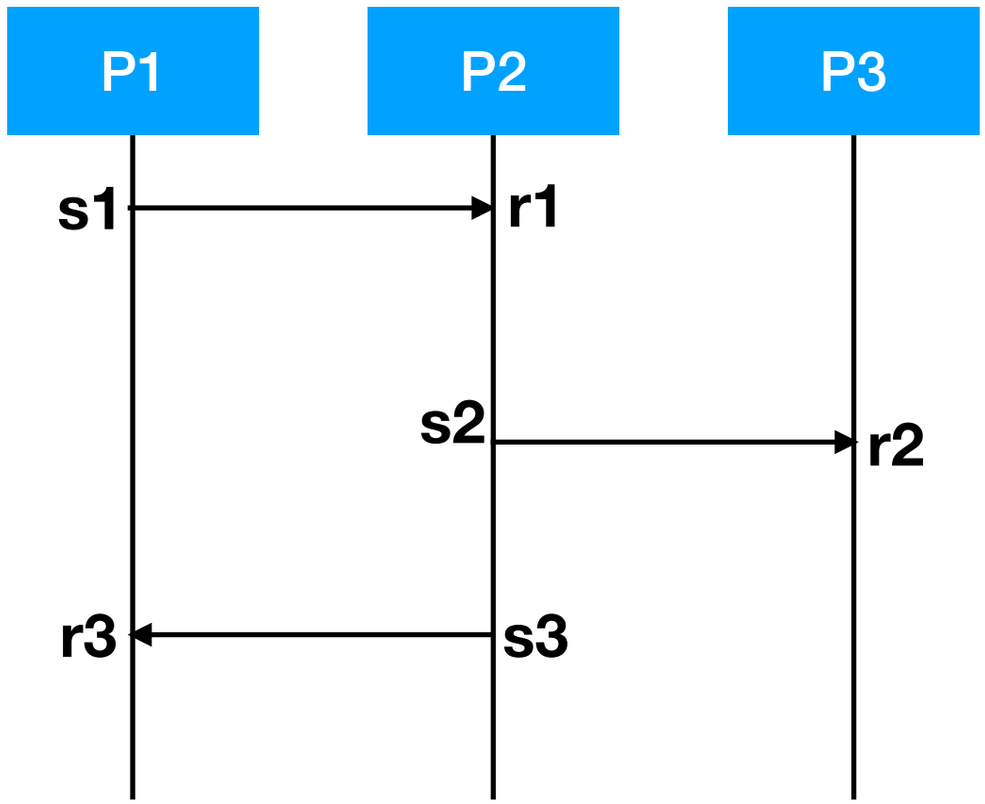
- **A finite set P of processes**
- **A finite set E of events, partitioned into S and R, for send and receive**
- **A function L : E -> P for labeling event ownership ; for some p, Ep is {e | L(e) = p}**
- **A bijective function C : S -> R for mapping send-receive pairs**
- **For some p, a relation <p for describing local visual order as indicated by MSC, and a relation < := (Up<p) U {(s, C(s)) | s of S}, the set of unions of all local visual orders and all send-receive pairs, for joining orders of processes and ensuring corresponding send is prior to the receive**

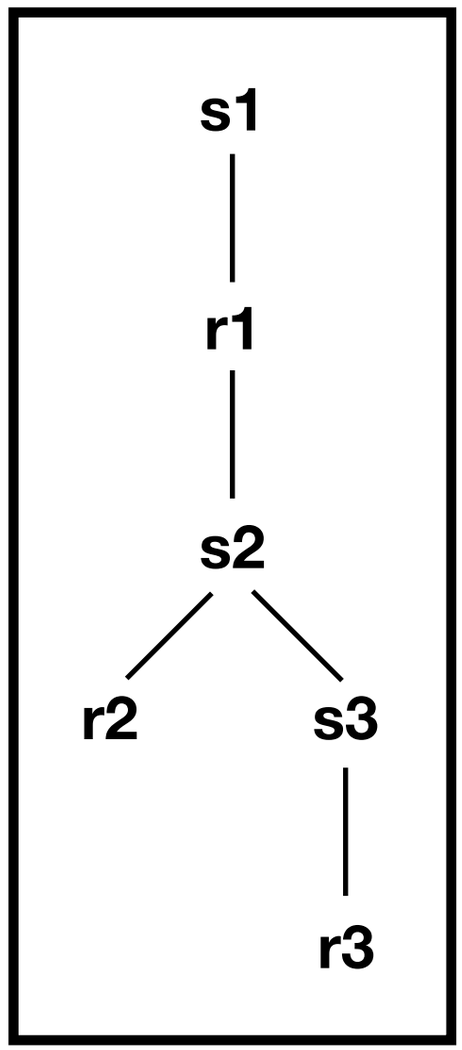


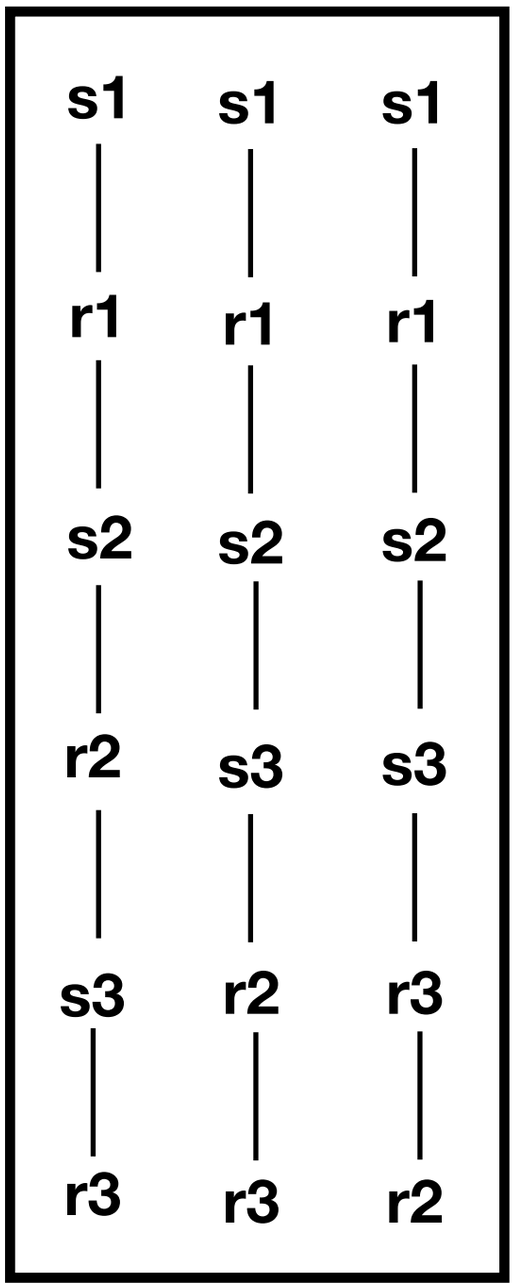
* **P = {P1,P2,P3}**
* **E = {s1,s2,s3,r1,r2,r3}**
* **S = {s1,s2,s3} R = {r1,r2,r3}**
* **L = {(s1,P1),(r3,P1),(r1,P2),(s2,P2),(s3,P2), (r2,P3)}**
* **C = {(s1,r1),(s2,r2),(s3,r3)}**
* **<p1 = {(s1,r3)} <p2 = {(r1,s2),(r1,s3),(s2,s3)} <p3 ={}**
* **< = {(s1,r3),(r1,s2),(r1,s3),(s2,s3),(s1,r1), (s2,r2),(s3,r3)}**

Formalism (Model Checking of Message Sequence Charts)

- **< is a partial order relation**
- **A linearization of a partial order relation is the linear extension of that, i.e., a total order relation such that it includes that partial order relation**
- **Hasse diagram, with transitive property represented from top to bottom**
- **< = {(s1,r3),(r1,s2),(r1,s3),(s2,s3),(s1,r1),(s2,r2),(s3,r3)}**



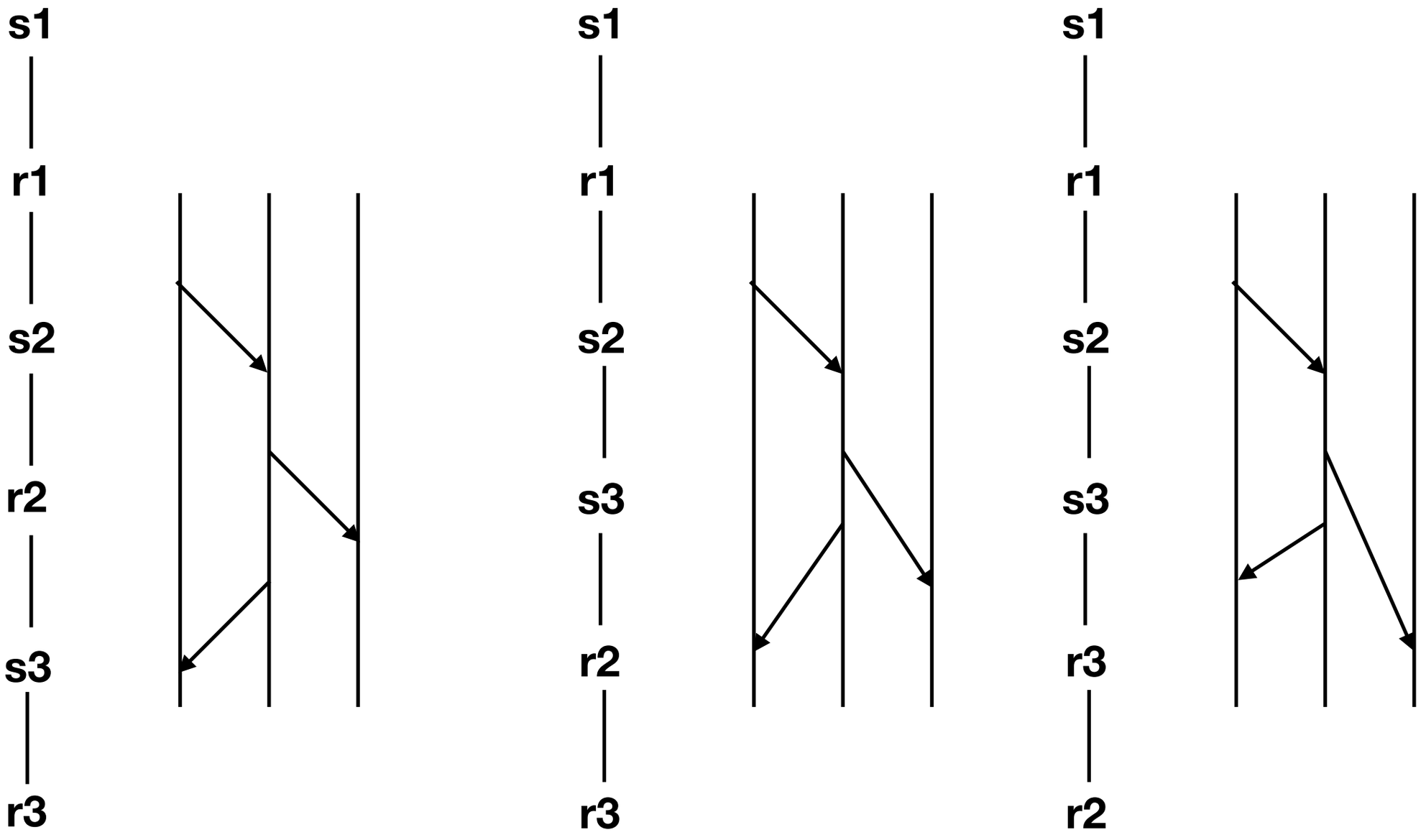




partial order: reflexivity, antisymmetry, transitivity
total order: relation: partial's + totality

Async behavior

- **Because the global < is defined by the union of local <p plus the pairwise send-before-receive constraint, global trace differs**
- **I surmise this interpretation of private locality is for the modeling of distributed concurrent system where processes share no knowledge of others' progresses**

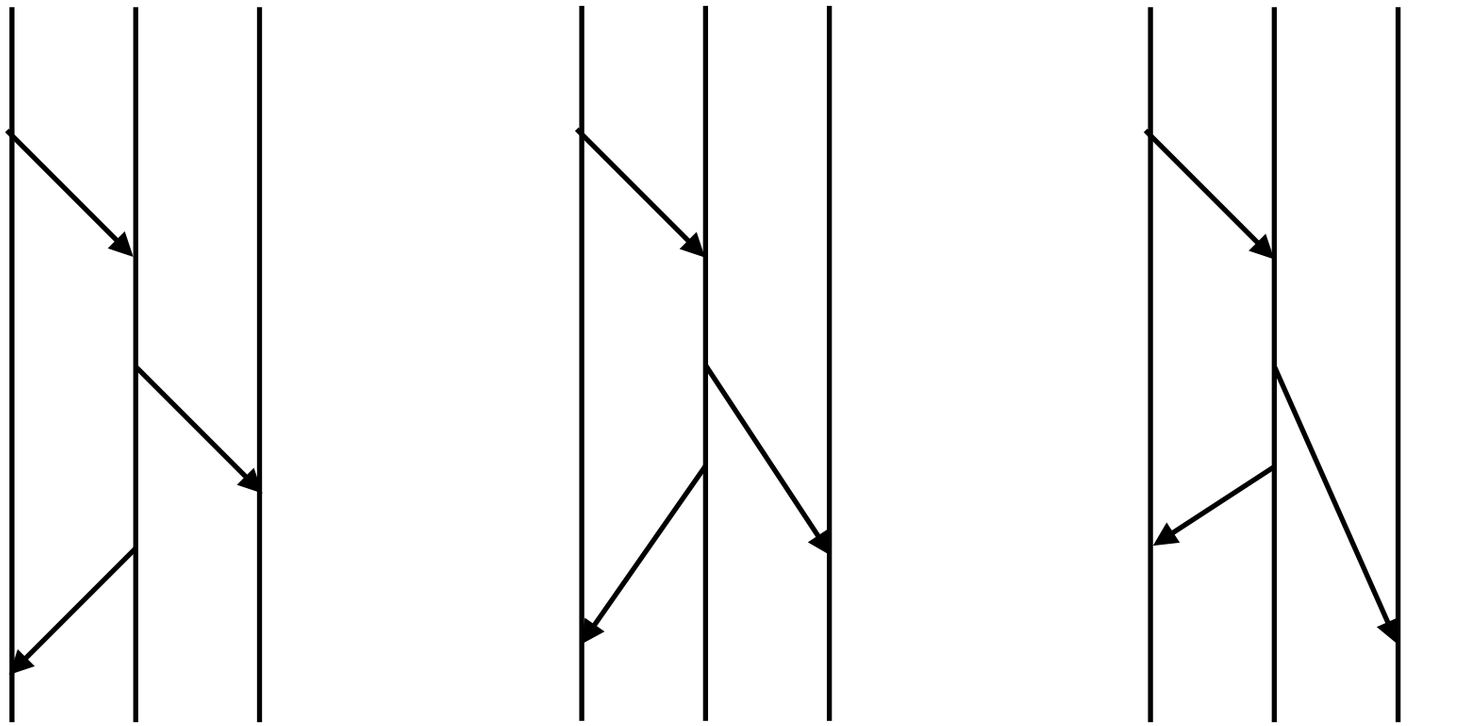


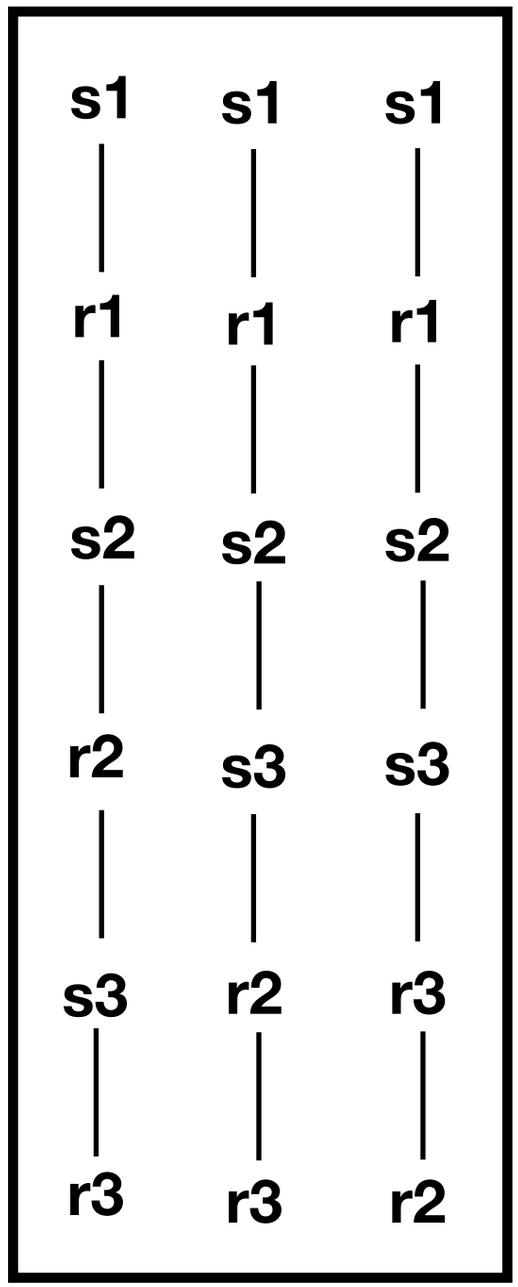
1. Unfinished business
    1. Angluin with Mealy
    2. Message synchronization
    3. Subject of learned model
2. **MSC**
    1. Formalism
    2. **Model extraction**
    3. Implementation problem
    4. Attempt at TCP
3. MSC-graph and HMSC
    1. Formalism
    2. Model extraction
    3. Attempt at TLS

Languages of MSCs

- **For a MSC M and an alphabet Σ, a Σ-labeled MSC is a pair (M, l) where l is a function E -> Σ that labels the events of M with an alphabet**
- **The language of a MSC M, denoted L(M), is the set of all possible linearization passed to labeling function l**

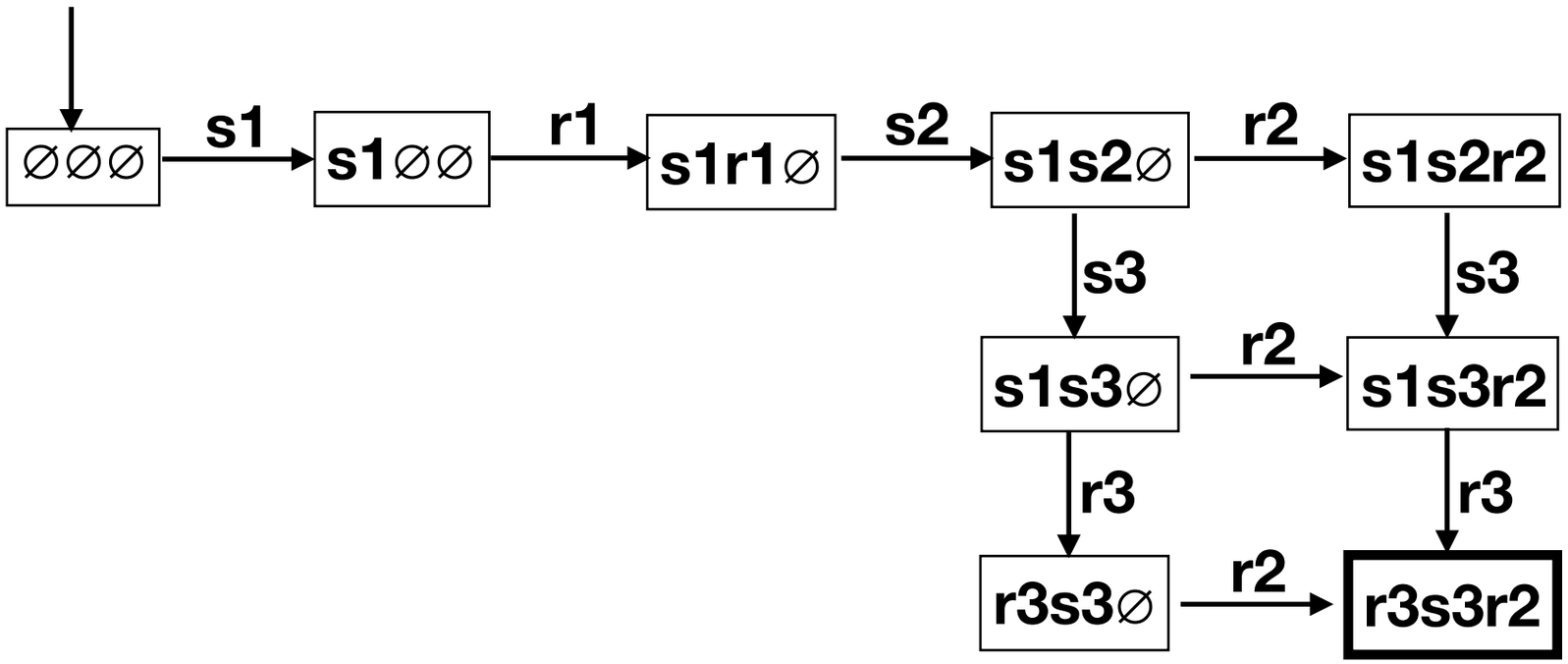
- **Suppose l maps each events in E to an symbol in Σ with the same name**
- **L(M) = {s1r1s2r2s3r3, s1r1s2s3r2r3, s1r1s2s3r3r2}, namely, the linearizations of partial order relation <**

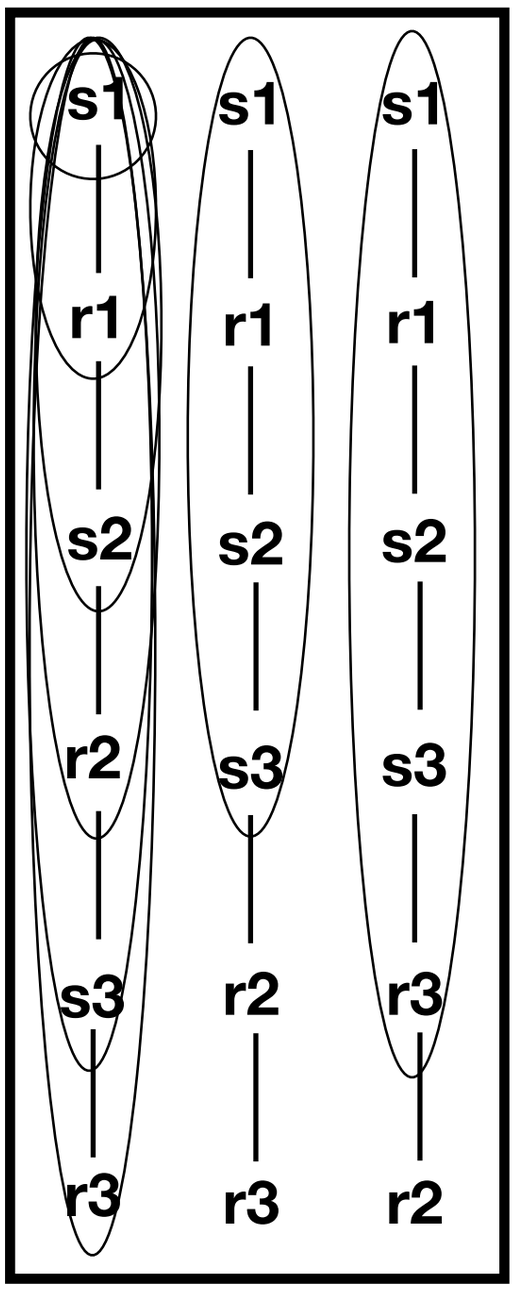




Automata of MSCs

- **A cut c is a subset of E such that if e in c and e' < e then e' in c,**
  c is closed with respect to < (?)
- **To construct an automaton AM accepting L(M):**
  1. **the cuts are the states, marked by max events of each processes**
  2. **the empty cut is the initial state**
  3. **the cut c = E is the final state**
  4. **if cut d is the union of cut c and singleton event e, there is a transition from c to d on l(e)**
- **Ep1 = {s1,r3} / Ep2 = {r1,s2,s3} / Ep3 ={r2}**





Model Checking of MSCs

- **To model-check a Σ-labeled MSC M:**
  1. **Represent constraints by an automaton A over Σ which accepts the undesirable executions**
  2. **Check if the intersection of L(M) and L(A) is empty by checking the language of the product automaton**

- **Given a Σ-labeled MSC M with n events and k processes, and an automaton A of size m, the model checking problem (M, A) can be solved in time O(m*n^k), and is coNP-complete**

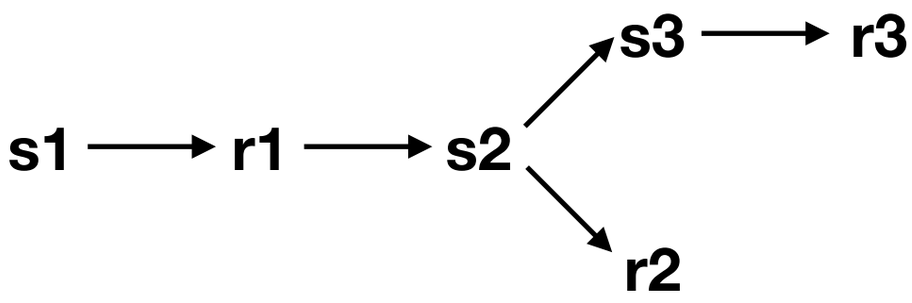
***Not exactly useful to us. Since our MSCs at hands of RFCs should be treated as specs themselves. We need only the model extraction part.**

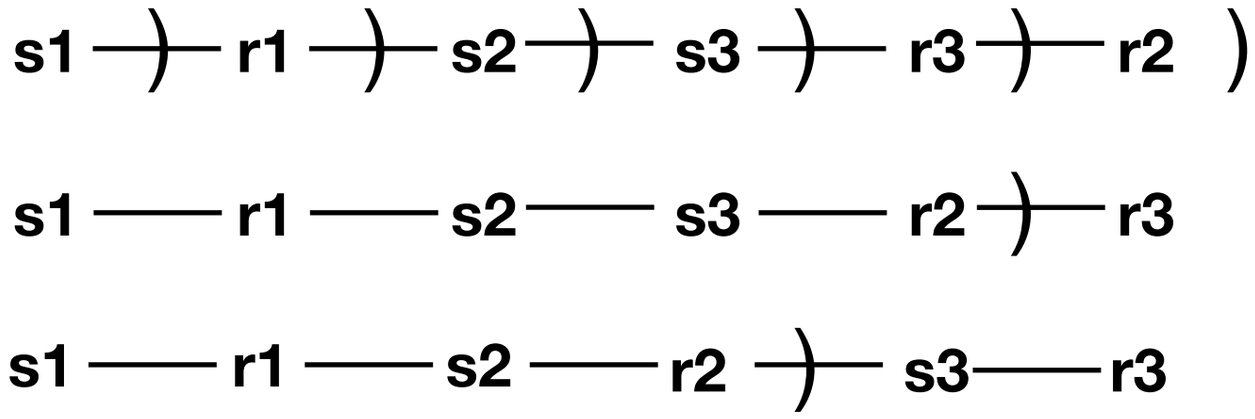
1. Unfinished business
    1. Angluin with Mealy
    2. Message synchronization
    3. Subject of learned model
2. **MSC**
    1. Formalism
    2. Model extraction
    3. **Implementation problem**
    4. Attempt at TCP
3. MSC-graph and HMSC
    1. Formalism
    2. Model extraction
    3. Attempt at TLS

Implementation Design

- **Problem 1: generating linearization. There are algorithms available:**
  1. **"Constant Time Generation of Linear Extensions**
  2. **"Generating Linear Extensions Fast"**
  3. **"A loop-free algorithm for generating the linear extensions of a poset"**
  4. **...**
- **But is it necessary to generate linearizations? Depends on how we extract cuts?**
- **But if we can generate linearizations, is it necessary to extract cuts?**
- **Since cuts are for generating a DFA, and all we wanted to do is to get L(M), which is exactly the finite set of the linearizations, so there's no need to actually get an automaton model? (plus we don't need it for model checking)**

- **Problem 2: extracting cuts. Is there algorithm?**
  Traverse Hasse diagrams as directed graphs? Check all prefixes of linearizations?





1. Unfinished business
    1. Angluin with Mealy
    2. Message synchronization
    3. Subject of learned model
2. **MSC**
    1. Formalism
    2. Model extraction
    3. Implementation problem
    4. **Attempt at TCP**
3. MSC-graph and HMSC
    1. Formalism
    2. Model extraction
    3. Attempt at TLS

TCP Connection Synchronization MSC

As specified in RFC 793

3-Way

A				B
CLOSED				LISTEN
SYN-SENT	-->	SYN	-->	SYN-RCVD
ESTABLISHED	<--	SYN+ACK	<--	SYN-RCVD
ESTABLISHED	-->	ACK	-->	ESTABLISHED

Ellipsis (...) indicates a segment which is still in the network (delayed).

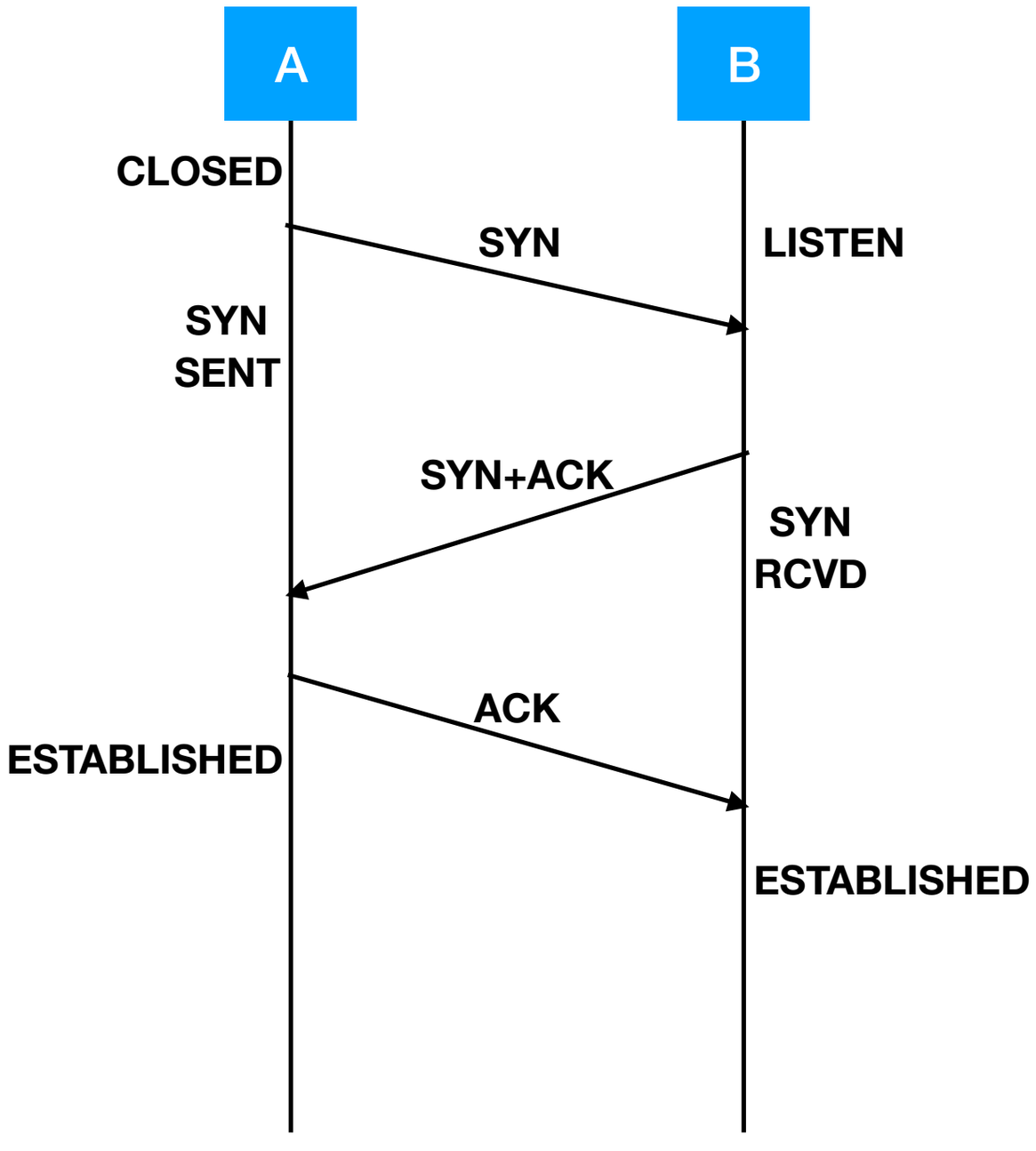
Simultaneous

A				B
CLOSED				CLOSED
SYN-SENT	-->	SYN	...	
SYN-RCVD	<--	SYN	<--	SYN-SENT
	...	SYN	-->	SYN-RCVD
SYN-RCVD	-->	SYN+ACK	...	
ESTABLISHED	<--	SYN+ACK	<--	SYN-RCVD
	...	ACK	-->	ESTABLISHED

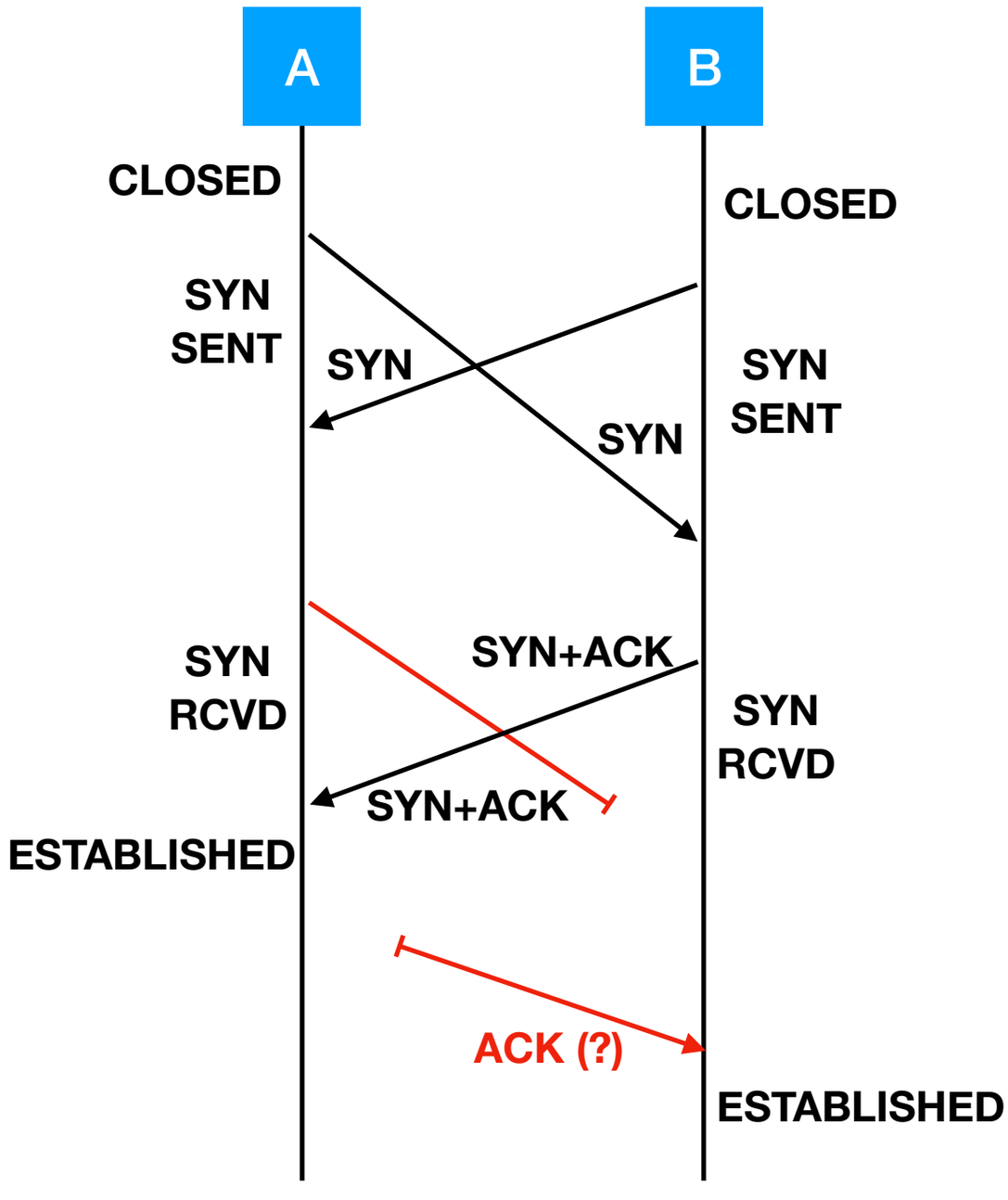
TCP Connection Synchronization MSC

- **Problem 1: TCP automaton is a local model while we can only extract global models**
- **Problem 2: how to map process states into some send/receive events?**
- **Problem 3: what to do with states irrelevant to message, like CLOSED and LISTEN?**
- **Problem 4: what to do with delayed message?**

3-Way



Simultaneous



TCP Connection Synchronization MSC

- **Problem 1: TCP automaton is a local model while we can only extract global models**
  1. **separate client and server side messages and states?**
  2. **construct a product automaton with appropriate start and final states?**

- **Problem 2: how to map process states into some send/receive events?**
  1. **encode messages as Cartesian product (message x state)?**

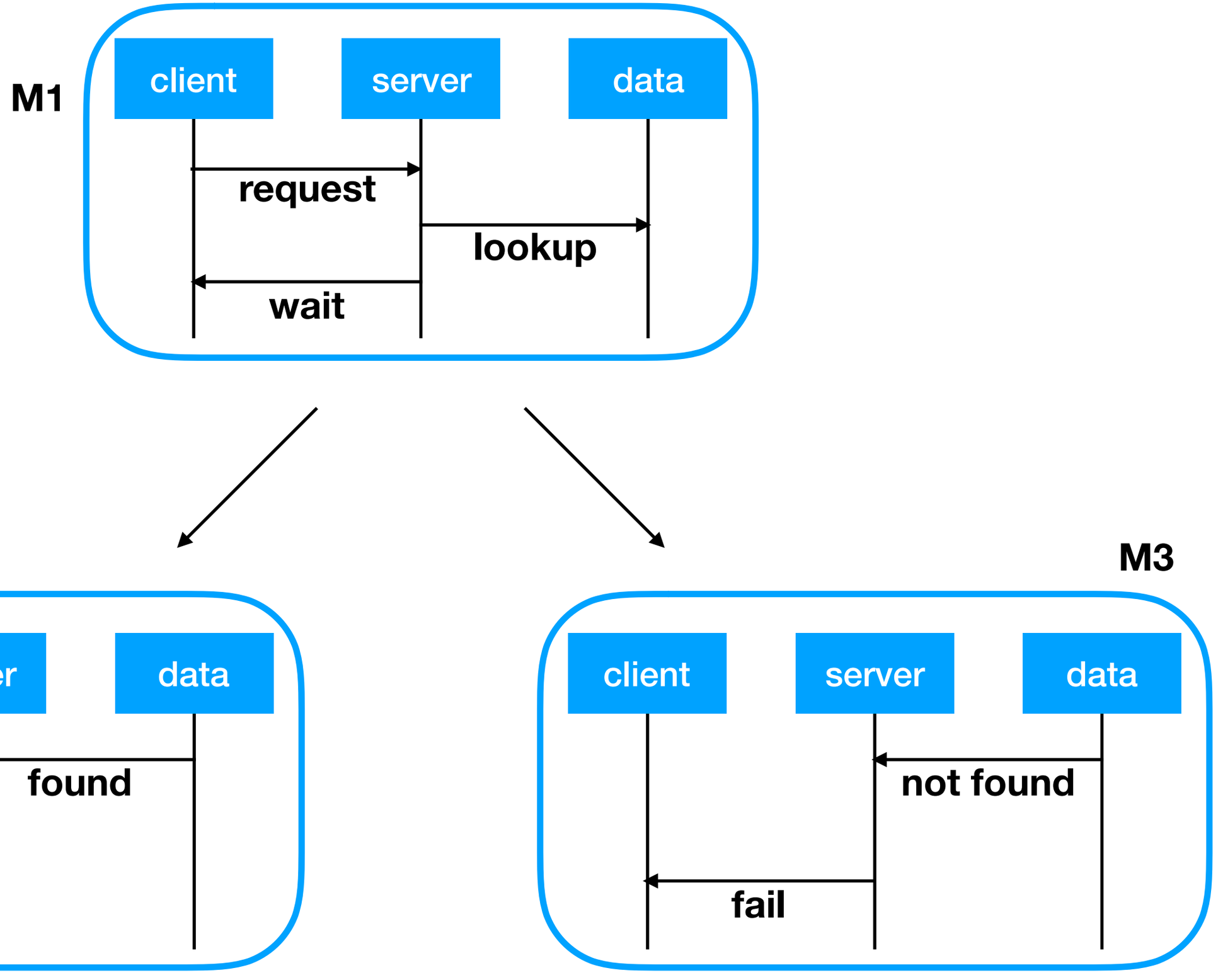
- **Problem 3: what to do with states irrelevant to message, like CLOSED and LISTEN**
  1. **ignore system call?**
  2. **add an additional process, environment, that sends these messages to them?**

- **Problem 4: what to do with delayed message?**
  1. **send them to and from an additional process, environment?**

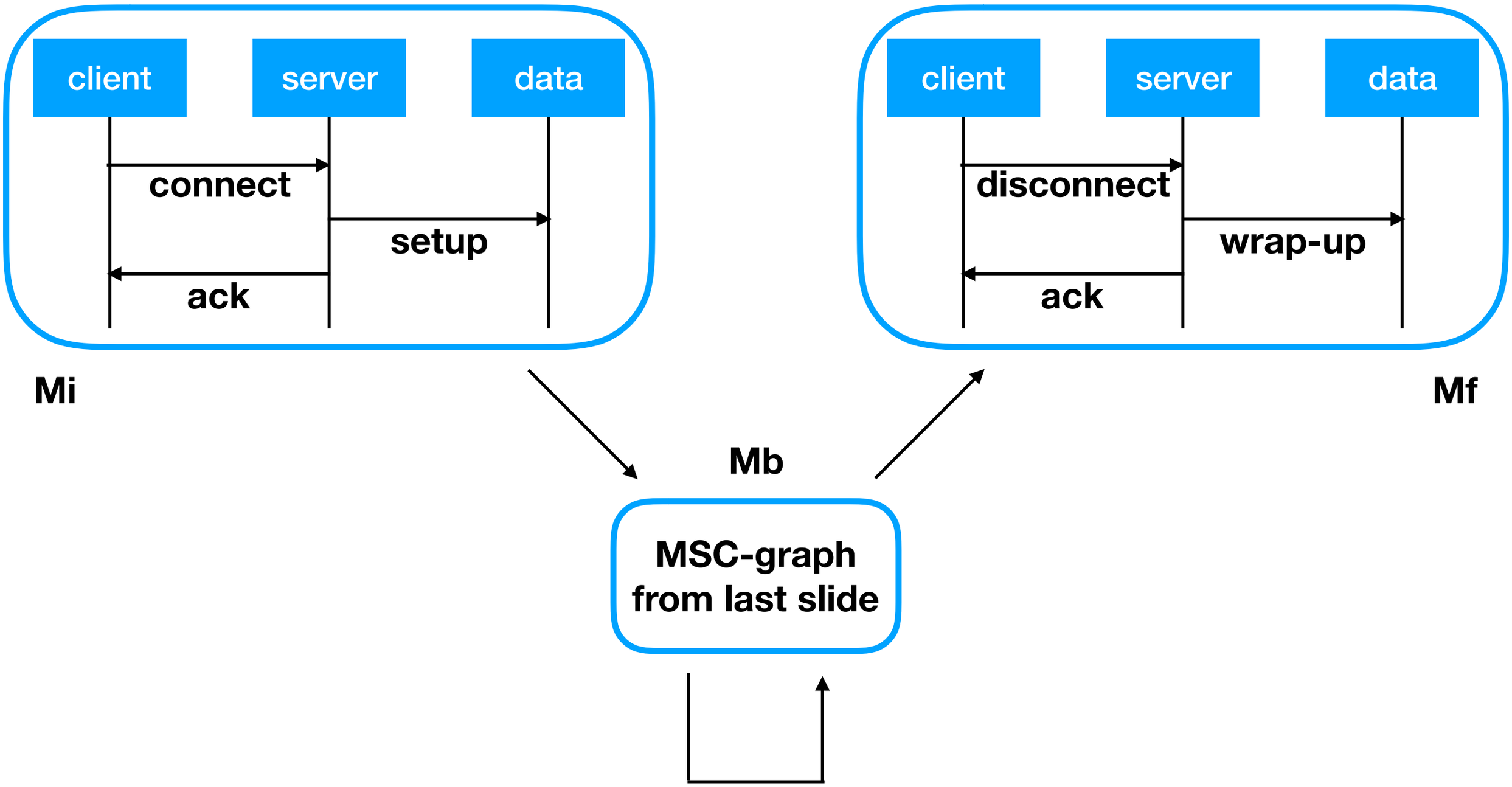
Would the resulting model be the correct model? And are these MSCs even enough?

1. Unfinished business
    1. Angluin with Mealy
    2. Message synchronization
    3. Subject of learned model
2. MSC
    1. Formalism
    2. Model extraction
    3. Implementation problem
    4. Attempt at TCP
3. **MSC-graph and HMSC**
    1. Formalism
    2. Model extraction
    3. Attempt at TLS

MSC-graph


M1
client
server
data
request
lookup
wait
M2
client
server
data
found
reply
M3
client
server
data
not found
fail

HMSC


client
server
data
connect
setup
ack
client
server
data
disconnect
wrap-up
ack
Mi
Mb
Mf
MSC-graph
from last slide

1. Unfinished business
    1. Angluin with Mealy
    2. Message synchronization
    3. Subject of learned model
2. MSC
    1. Formalism
    2. Model extraction
    3. Implementation problem
    4. Attempt at TCP
3. **MSC-graph and HMSC**
    1. **Formalism**
    2. Model extraction
    3. Attempt at TLS

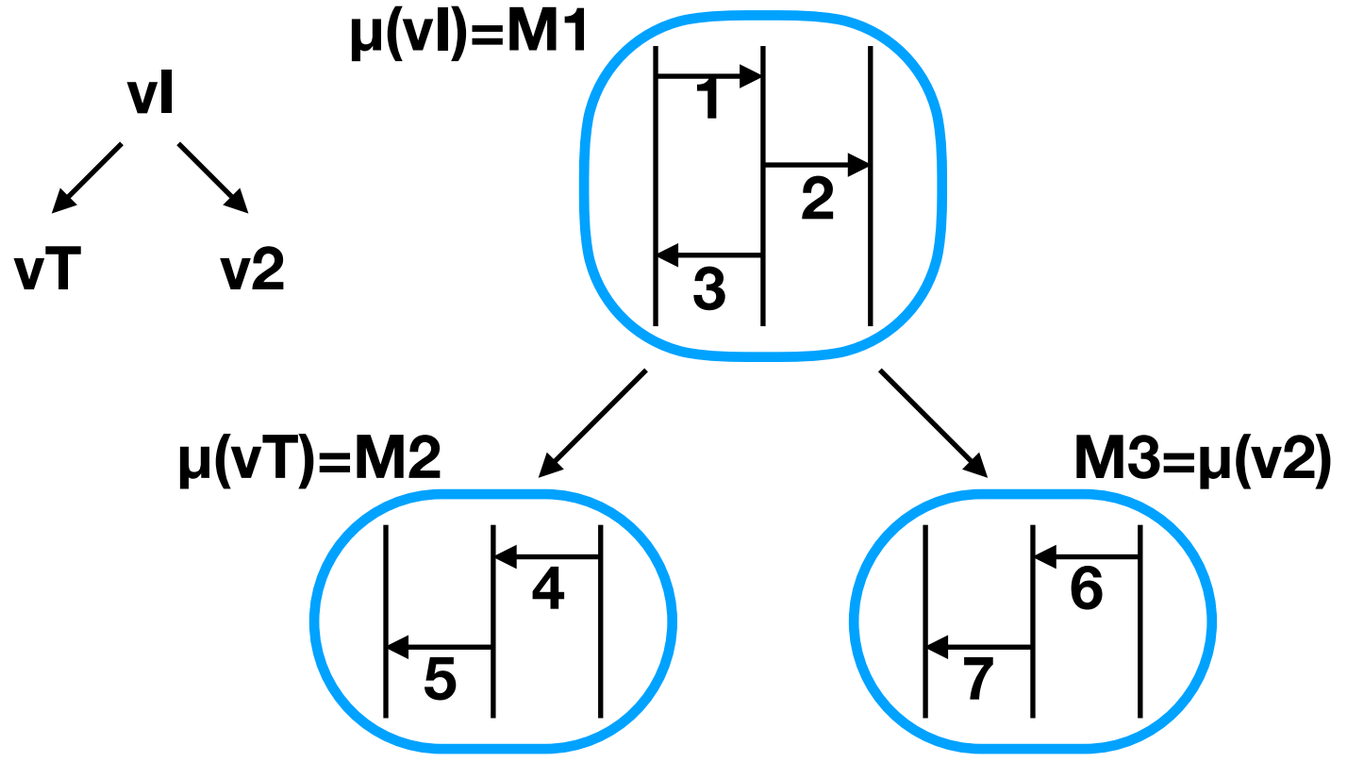
Formalism (Model Checking of Message Sequence Charts)

A MSC-graph G consists of:

- **A set of vertices V**
- **A binary relation → over V**
- **An initial vertex vI**
- **A terminal vertex vT**
- **A labeling function μ : V -> MSC**

A Hierarchical MSC H consists of:

- **A finite set of nodes N**
- **A finite set of boxes(supernodes) B**
- **An initial node/box vI of NUB**
- **A terminal node/box vT of NUB**
- **A labeling function μ : (N -> MSC) or (B -> HMSC)**
- **A set of edges E, which is a subset of (NUB)x(NUB), that connects nodes and boxes**

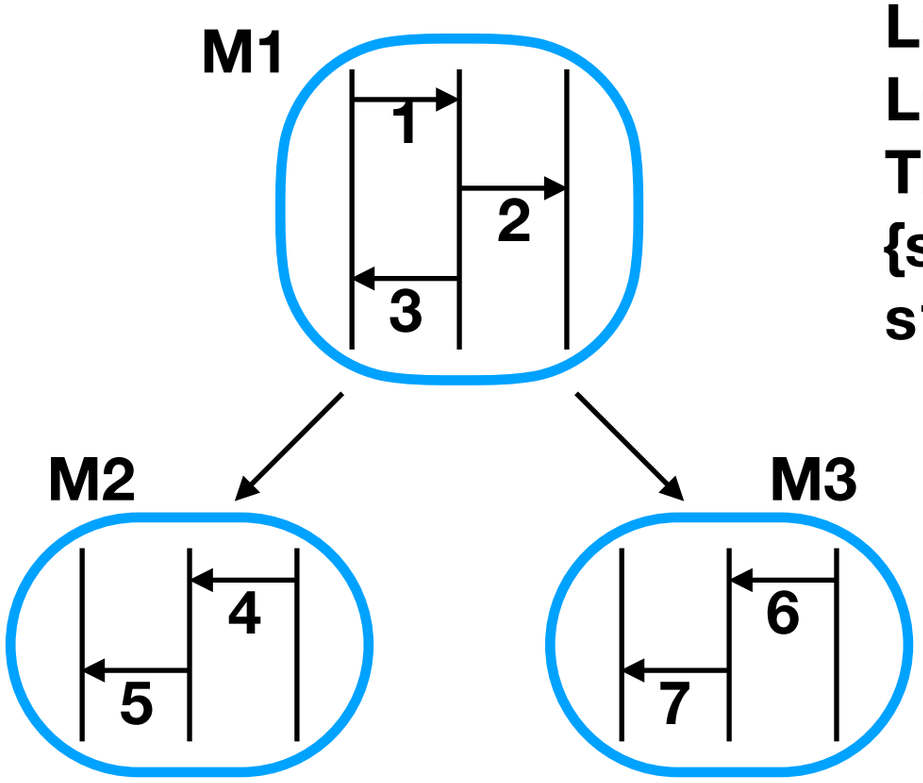


***A HMSC can be reduced to a MSC-graph by "flattening" it, by recursively substituting all b of B by μ(b) and revising the result to preserve the original properties.**
Thus, algorithms designed for MSC-graphs also work for HMSCs.

Concatenation of MSCs

In a MSC-graph G, linked MSCs are concatenated. Consider two vertex, v1 and v2, and their MSCs, M1 and M2, i.e., μ(v1) = M1 and μ(v2) = M2, if there is an edge (v1,v2) in →, there are two interpretation of concatenation:

- **Synchronous: all the events in M1 precede all the events in M2**
- **Asynchronous: "concatenating the two MSCs process by process" (?)**



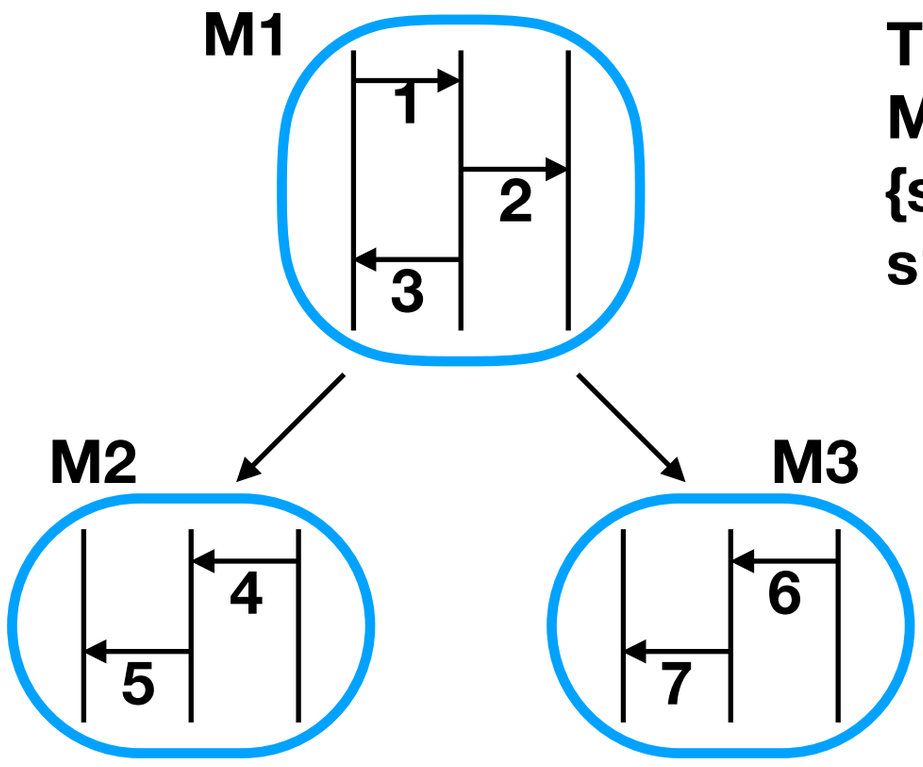
L(M1) = {s1r1s2r2s3r3, s1r1s2s3r2r3, s1r1s2s3r3r2}
L(M2) = {s4r4s5r5}
Their concatenation is simply L(M1) . L(M2) = {s1r1s2r2s3r3s4r4s5r5, s1r1s2s3r2r3s4r4s5r5, s1r1s2s3r3r2s4r4s5r5}

***Since I don't understand the async interpretation, I'll only consider the sync interpretation.**

1. Unfinished business
    1. Angluin with Mealy
    2. Message synchronization
    3. Subject of learned model
2. MSC
    1. Formalism
    2. Model extraction
    3. Implementation problem
    4. Attempt at TCP
3. **MSC-graph and HMSC**
    1. Formalism
    2. **Model extraction**
    3. Attempt at TLS

Languages of MSC-graphs

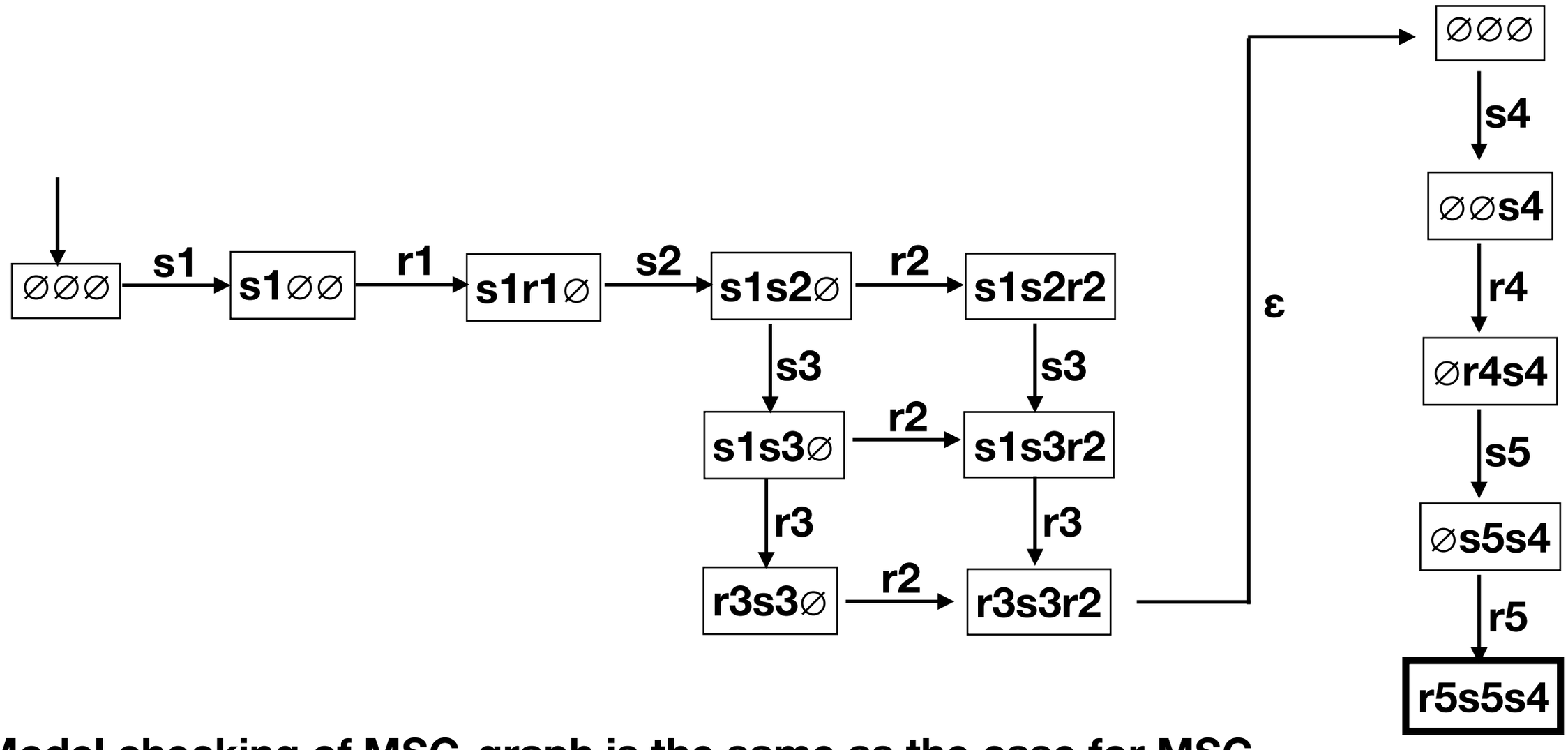
- **The language of two concatenated MSCs is the concatenation of the languages of the two MSCs.**
- **In a Σ-labeled MSC-graph, each vertex is mapped to a Σ-labeled MSC.**
- **A path is a sequence of vertices of V. An accepting path is a path from vI to vT.**
- **The language of a MSC-graph G, denoted L(G), is the set of strings of the form σ0σ1σ2 ... σn, where there exists an accepting path v0v1v2 ... vn, such that σi is in L(μ(vi). L(G) is the union of the concatenations of the languages of all the vertices on a path for all accepting paths, i.e., L(G) = union(for all p of accepting paths, concat(for all v of the vertices of p) L(v)).**



The only accepting path is vIvT, which μ maps to M1 and M2; thus, L(G) = L(M1) . L(M2) = {s1r1s2r2s3r3s4r4s5r5, s1r1s2s3r2r3s4r4s5r5, s1r1s2s3r3r2s4r4s5r5}

Automata of MSC-graphs

- **To construct an automaton AG accepting L(G):**
  1. **Replace each node v of G by the automaton Aμ(v) that accepts the language of μ(v)**
  2. **Standard automata concatenation operations: if there is an edge from vertex u to vertex v, add to all the final states in Aμ(u) an epsilon transition to the start state of Aμ(v); make the final states of the second automaton the final states of the combined automaton**



***Model checking of MSC-graph is the same as the case for MSC**

1. Unfinished business
    1. Angluin with Mealy
    2. Message synchronization
    3. Subject of learned model
2. MSC
    1. Formalism
    2. Model extraction
    3. Implementation problem
    4. Attempt at TCP
3. **MSC-graph and HMSC**
    1. Formalism
    2. Model extraction
    3. **Attempt at TLS**

TLS Full Handshake MSC

As specified in RFC 5246

Client		Server
ClientHello	-------->	
		ServerHello
		Certificate*
	<--------	ServerKeyExchange*
		CertificateRequest*
		ServerHelloDone
Certificate*		
ClientKeyExchange		
CertificateVerify*	-------->	
[ChangeCipherSpec]		
Finished		
		[ChangeCipherSpec]
	<--------	Finished

TLS Abbreviated Handshake MSC

As specified in RFC 5246

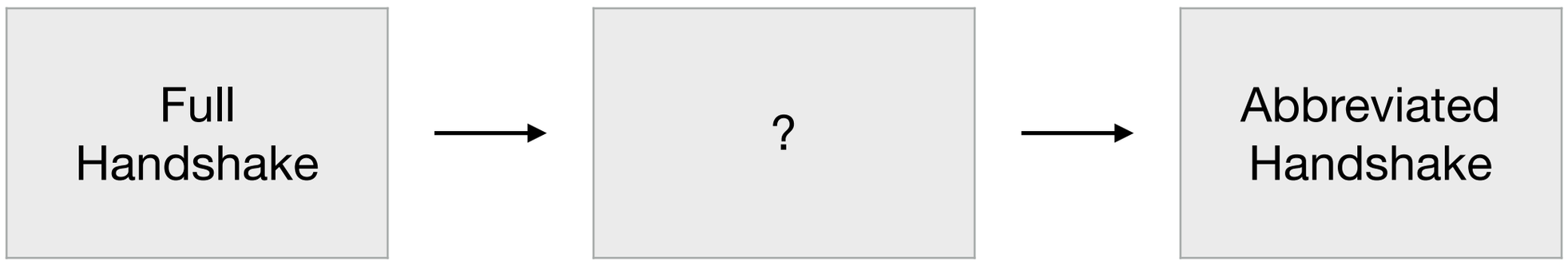
Per RFC, the abbreviated handshake protocol is used "when the client and server decide to resume a previous session or duplicate an existing session (instead of negotiating new security parameters)". So it should be modeled with a MSC-graph with the abbreviated handshake following the full handshake.

Client		Server
ClientHello	-------->	
	<--------	ServerHello
		[ChangeCipherSpec]
		Finished
[ChangeCipherSpec]	-------->	
Finished		

***Apparently [ChangeCipherSpec] is itself a protocol, which I suppose should be modeled with HMSC, but since I don't understand Hierarchical Kripke structure, so for now I'll take it as an atomic message.**

TLS Handshake MSC-graph

- **Problem 1: what to put between the two handshake?**



- **Problem 2: what to do with the optional part, like client authentication?**

  Per RFC, there are "optional or situation-dependent messages that are not always sent", which means we might need to break up the full handshake MSC.

  > **(F.1.1. Authentication and Key Exchange) TLS supports three authentication modes: authentication of both parties, server authentication with an unauthenticated client, and total anonymity.**

- **Problem 3: the message exchange sequence might depend on the cipher suite they choose, which means we might need to encode the message content and extend the MSC-graph with register, as they did in "A Messy State of the Union" In "Protocol State Fuzzing", their abstraction of message accounts for configs.**

THE END