The Real/Ideal Paradigm

Lecture 4

Alley Stoughton

Boston University

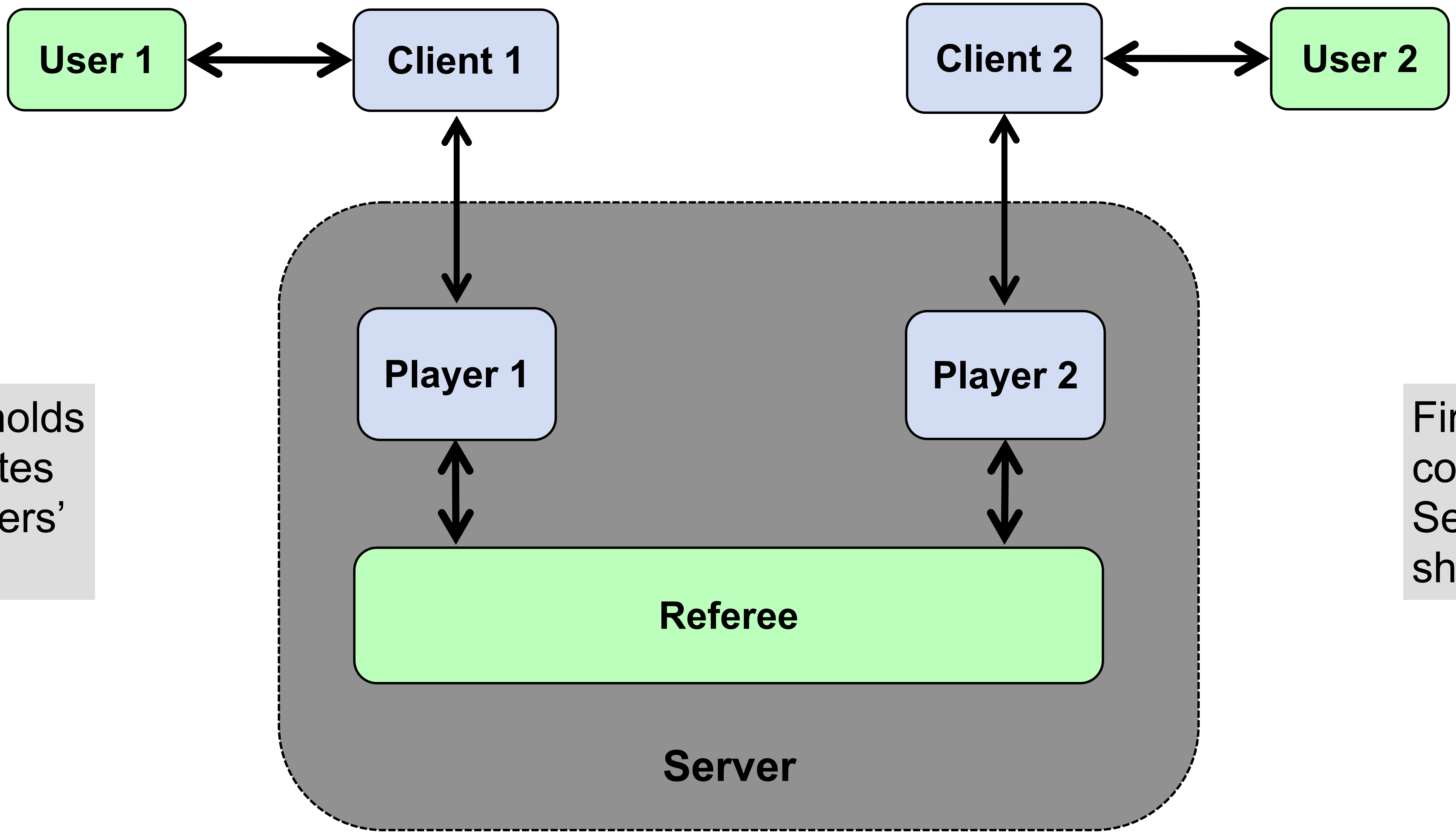
Oregon Programming Languages Summer School
June 3–13, 2024
Boston University

Example 3: Battleship (Review)

- We'll start this last lecture with a review of:
  - the program architecture of our secure battleship implementations in Haskell/LIO and Concurrent ML
  - our Real/Ideal Paradigm definition of security against a malicious player interface
- Then we'll survey the two implementations and consider how we used our security definition to audit them

Program Architecture and Behavior

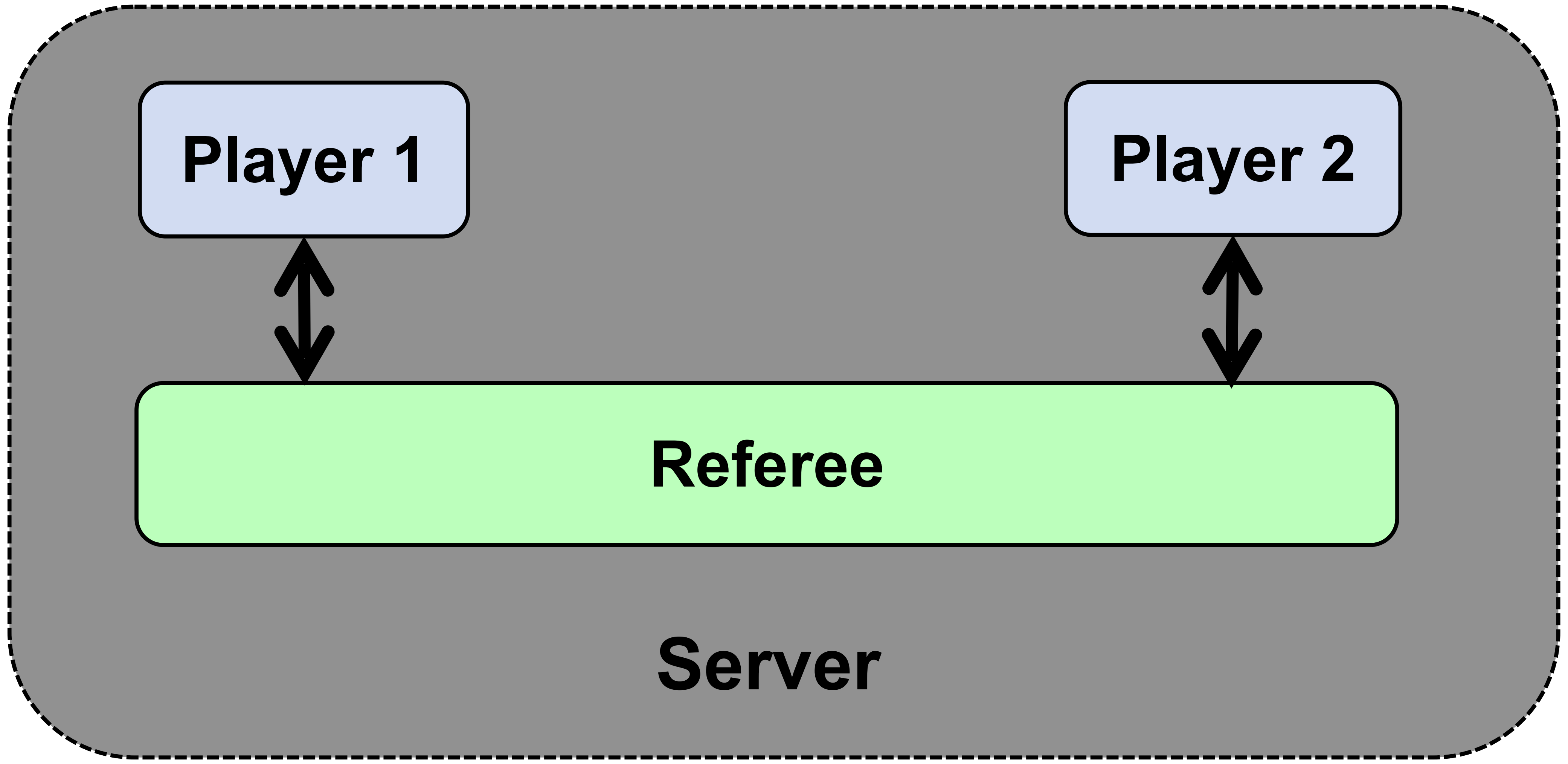
Referee holds and updates both Players' boards



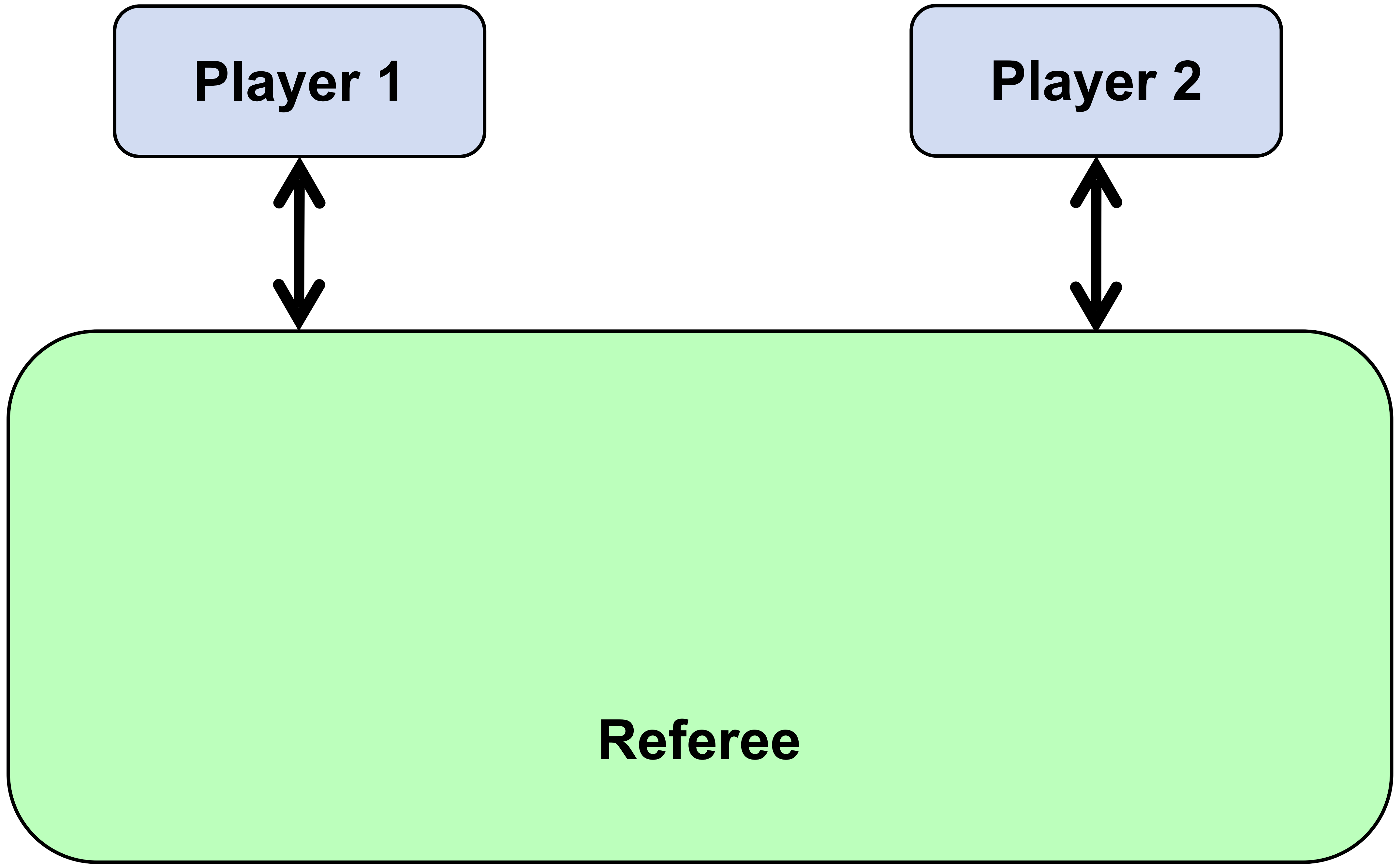
First Client to connect to Server gets to shoot first

Trusted Referee

- We implemented in Concurrent ML a trusted referee that holds and updates both player's boards, enforcing the rules of the game
- But we were also interested in reducing the trusted computing base (TCB), by splitting the referee into mutually distrustful *player interfaces*

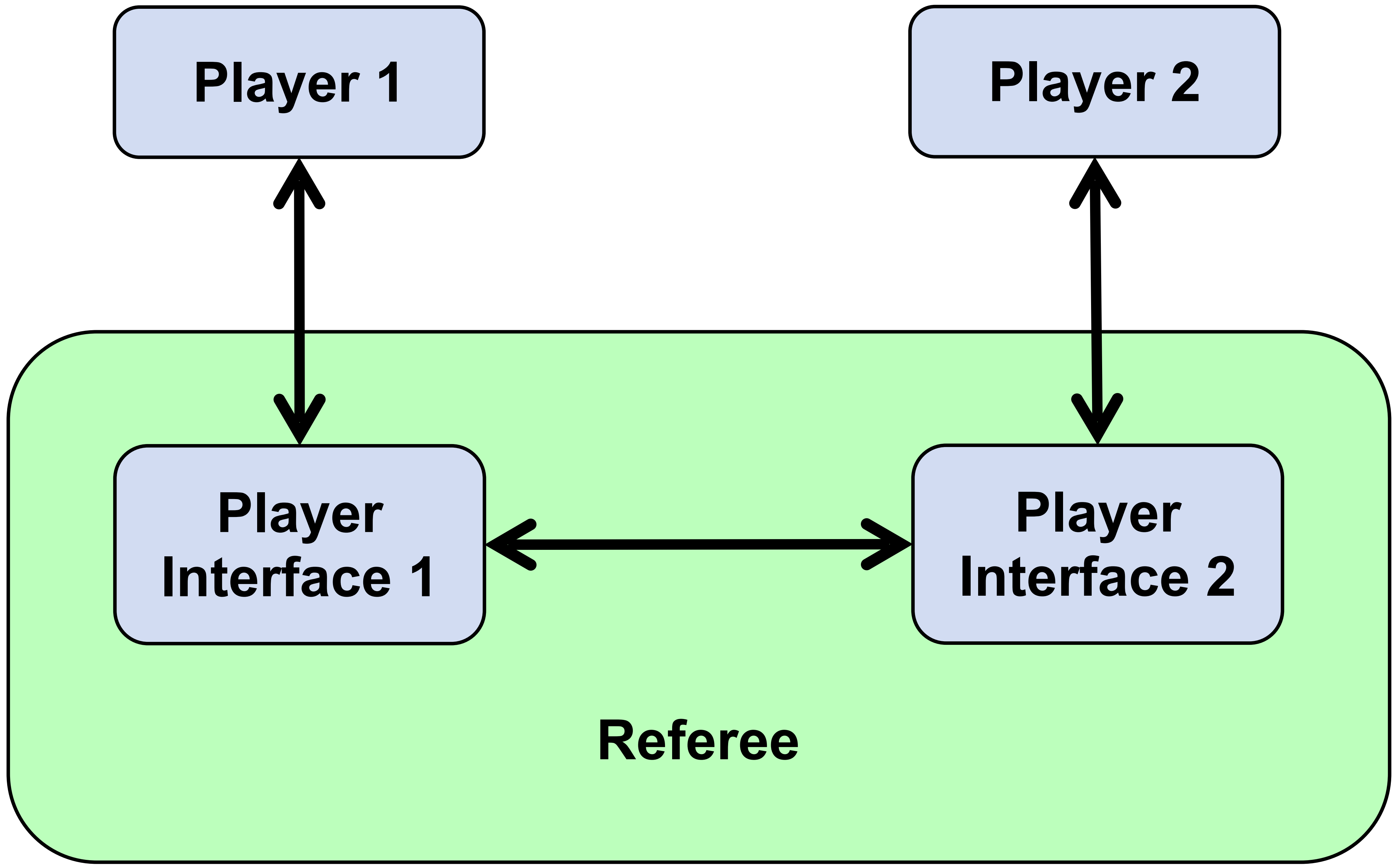


Splitting Referee into Mutually Distrustful Player Interfaces (PIs)



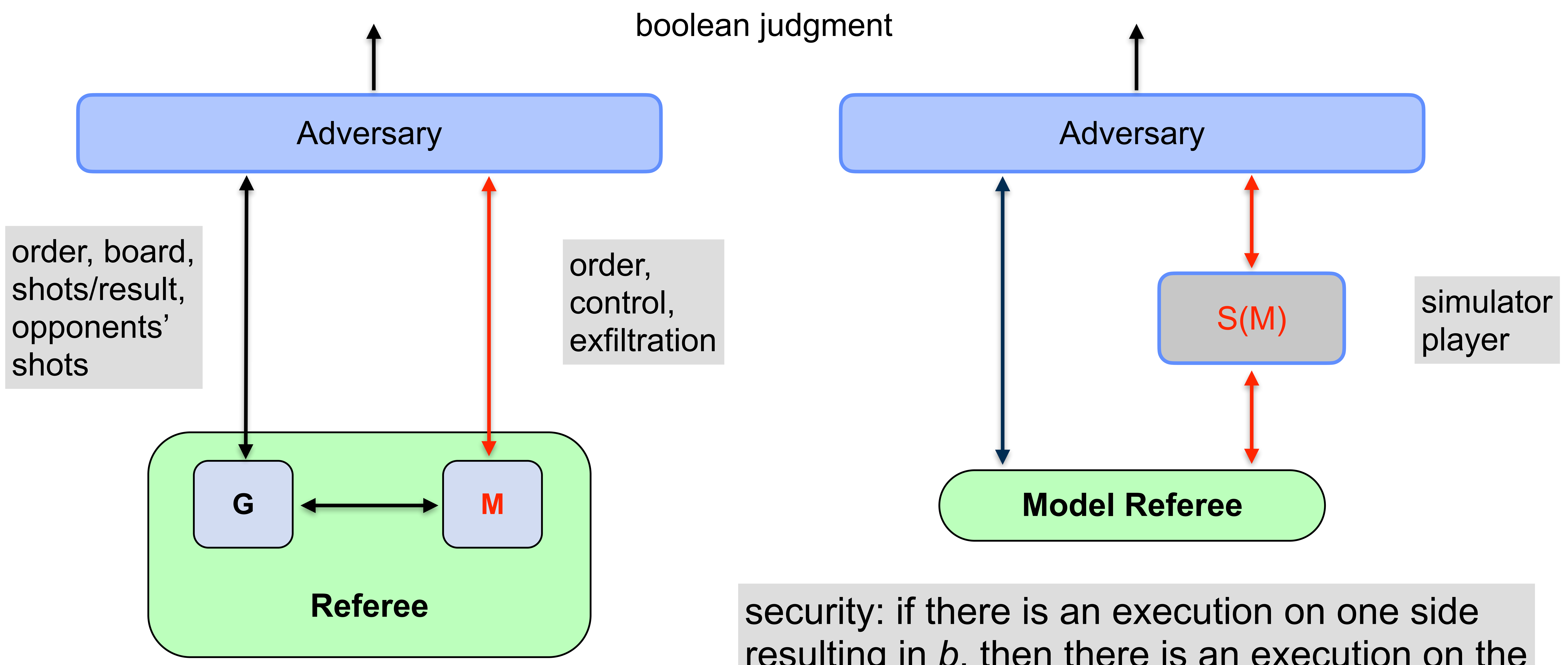
Splitting Referee into Mutually Distrustful Player Interfaces (PIs)

PIs will rely on some *trusted infrastructure*



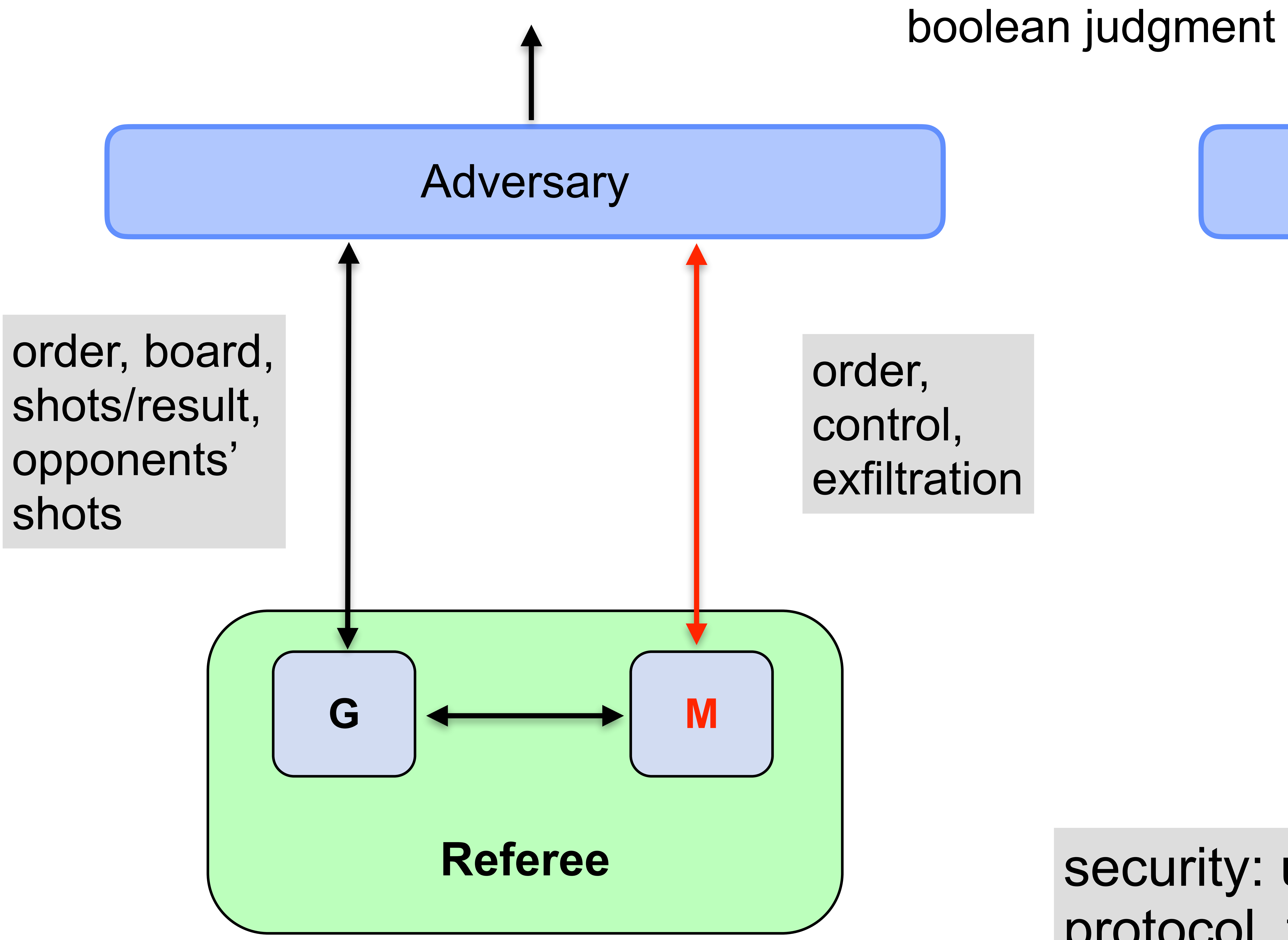
How do we define security against a malicious opponent PI?

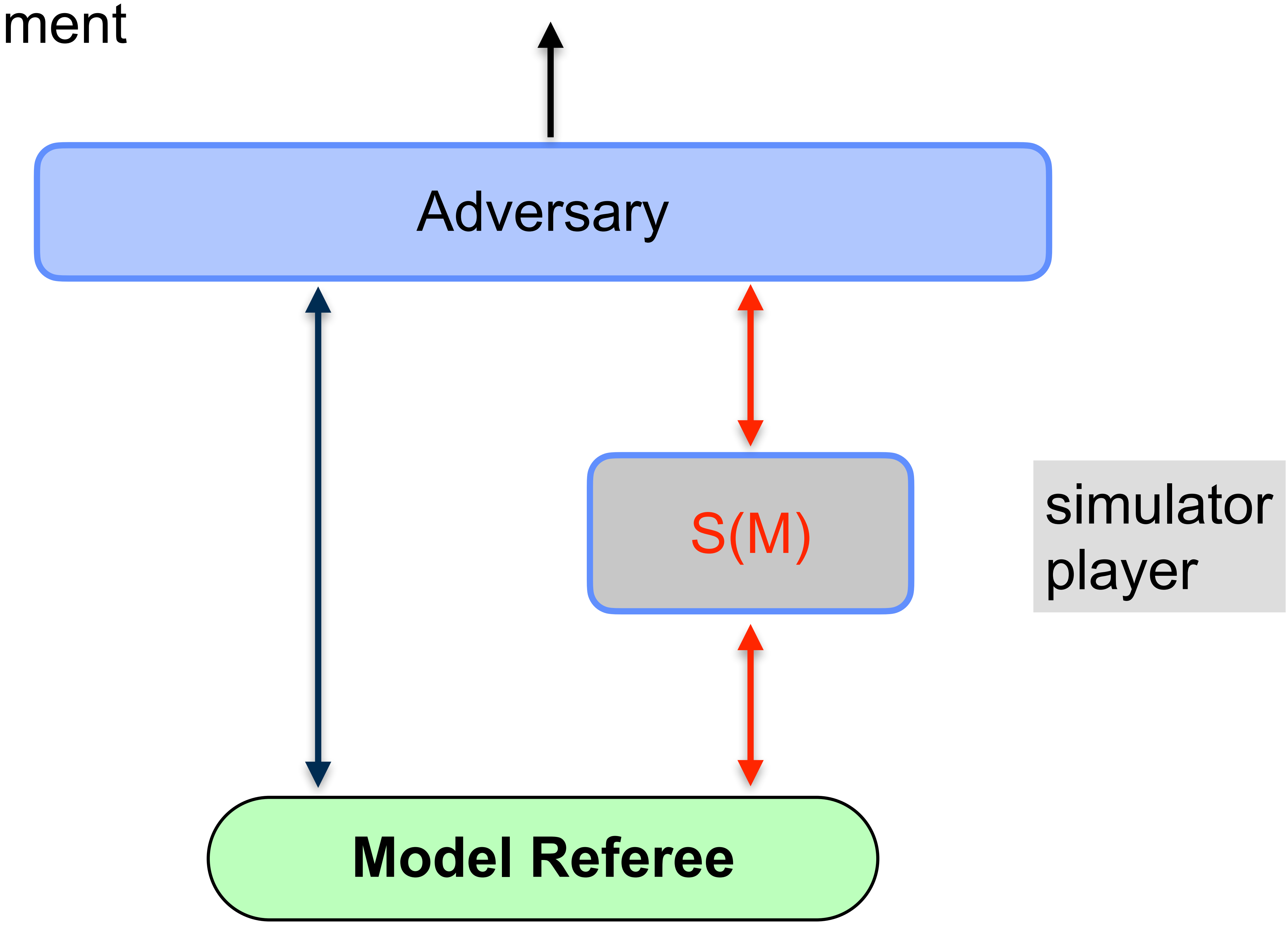
Security Against Malicious PI (Tentative)



security: if there is an execution on one side resulting in *b*, then there is an execution on the other side also resulting in *b*

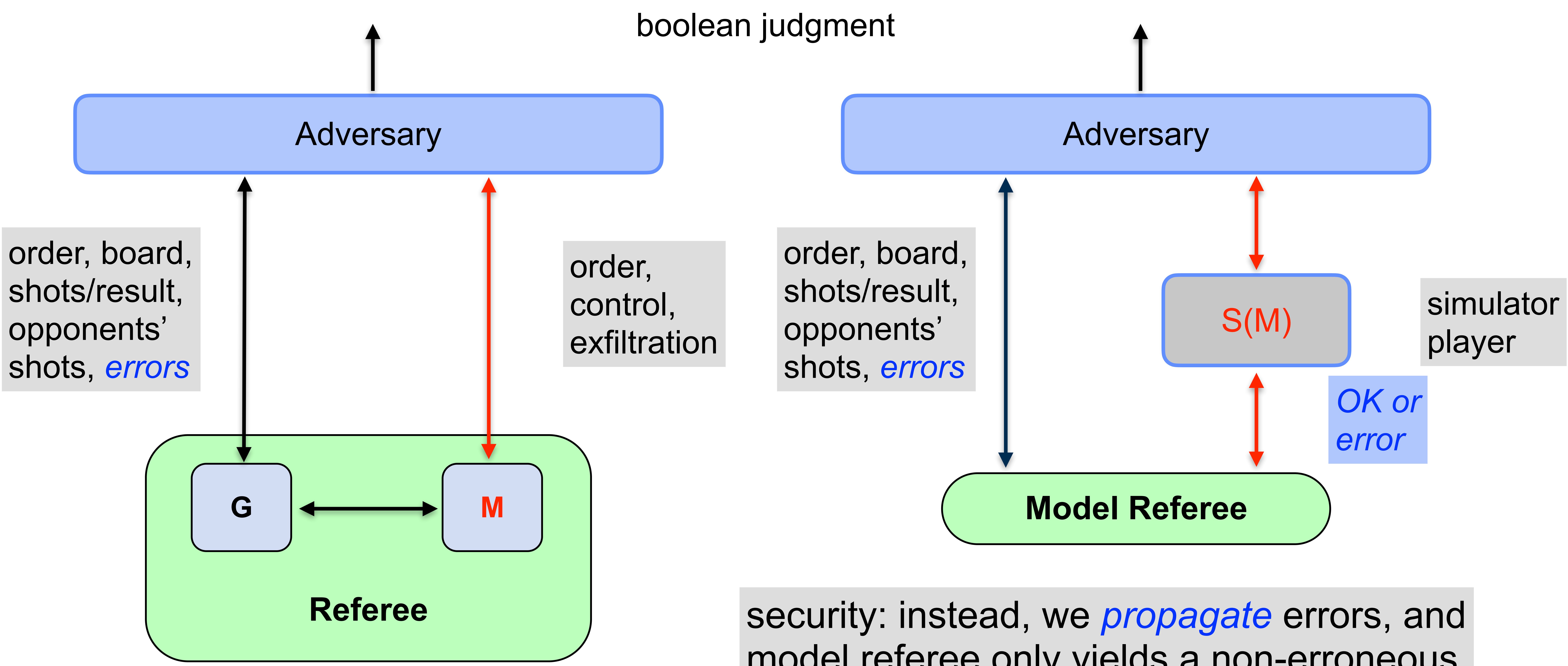
Security Against Malicious PI (Tentative)





security: unfortunately, if M doesn't follow the protocol, the error behavior (*termination*) in the two worlds can be different

Security Against Malicious PI



security: instead, we *propagate* errors, and model referee only yields a non-erroneous result if simulator player says OK

Pointers to Paper and Code

- On GitHub

https://github.com/alleystoughton/battleship

you can find a link to our PLAS 2014 paper *You Sank My Battleship!: A Case Study in Secure Programming* plus the Haskell/LIO and Concurrent ML code

- Note that the error propagation presented above is *not* followed by this code or described in the paper

Ambiguity Example: Patrol Boat

	A	B	C	D	E	F	G	H	I	J
A										
B						b				
C	c	c	c	c	c	b				
D						b				
E						b				
F										
G			p	s	s	s				
H			p				d			
I							d			
J							d			

GD
HC
GC

Ambiguity Example: Patrol Boat

	A	B	C	D	E	F	G	H	I	J
A										
B						b				
C	c	c	c	c	c	b				
D						b				
E						b				
F										
G			p	p						
H			s				d			
I			s				d			
J			s				d			

GD
HC
GC

LIO

- LIO is a library for Concurrent Haskell with dynamic enforcement of information flow control
- Information flow labels have both secrecy and integrety components
- Provides mutable variables, which can be shared between threads, and used for communication

LIO Battleship

- PIs exchange — using trusted code — labeled boards, made of labeled cells:

```
data LSR =  -- labeled shot result
       Miss        -- a miss
     | Hit         -- hit an unspecified ship
     | Sank Ship   -- sank a specified ship

data LC =  -- labeled cell
  LC
  (DCLabeled
   (Principal,  -- originating player interface
    Principal,  -- receiving player interface
    Pos,        -- position of cell
    DC LSR      -- DC action for shooting cell
   ))
```

LIO Example

PI 1

PI 2

Patrol Boat
MVar

LIO Example

PI 1

PI 2

1 : (1, 2, GC, pb) : 1 $\wedge$ 2

1 : (1, 2, HC, pb) : 1 $\wedge$ 2

Patrol Boat
MVar

LIO Example

PI 1

1 : (1, 2, HC, pb) : 1 ∧ 2

PI 2

1 : (1, 2, GC, pb) : 1 ∧ 2

1 : (1, 2, HC, pb) : 1 ∧ 2

Patrol Boat MVar

LIO Example

PI 1

: (1, 2, HC, pb) : 1 ∧ 2

PI 2

1 : (1, 2, GC, pb) : 1 ∧ 2

1 : (1, 2, HC, pb) : 1 ∧ 2

Patrol Boat MVar

LIO Example

PI 1

: (1, 2, HC, pb) : 1 $\wedge$ 2

PI 2

1 : (1, 2, GC, pb) : 1 $\wedge$ 2

1 : (1, 2, HC, pb) : 1 $\wedge$ 2

Patrol Boat MVar

HC

LIO Example

PI 1

: (1, 2, HC, pb) : 1 ∧ 2

PI 2

1 : (1, 2, GC, pb) : 1 ∧ 2

1 : (1, 2, HC, pb) : 1 ∧ 2

: (1, 2, HC, pb) : 1 ∧ 2

Patrol Boat MVar

HC

LIO Example

PI 1

: (1, 2, HC, pb) : 1 ∧ 2

PI 2

1 : (1, 2, GC, pb) : 1 ∧ 2

1 : (1, 2, HC, pb) : 1 ∧ 2

Yields Hit

: (1, 2, HC, pb) : 1 ∧ 2

Patrol Boat
MVar

HC

LIO Example

PI 1

: (1, 2, HC, pb) : 1 ∧ 2

1 : (1, 2, GC, pb) : 1 ∧ 2

PI 2

1 : (1, 2, GC, pb) : 1 ∧ 2

1 : (1, 2, HC, pb) : 1 ∧ 2

Yields Hit

: (1, 2, HC, pb) : 1 ∧ 2

Patrol Boat MVar

HC

LIO Example

PI 1

: (1, 2, HC, pb) : 1 ∧ 2

: (1, 2, GC, pb) : 1 ∧ 2

PI 2

1 : (1, 2, GC, pb) : 1 ∧ 2

1 : (1, 2, HC, pb) : 1 ∧ 2

Yields Hit

: (1, 2, HC, pb) : 1 ∧ 2

Patrol Boat
MVar

HC

LIO Example

PI 1

: (1, 2, HC, pb) : 1 ∧ 2

: (1, 2, GC, pb) : 1 ∧ 2

PI 2

1 : (1, 2, GC, pb) : 1 ∧ 2

1 : (1, 2, HC, pb) : 1 ∧ 2

Yields Hit

: (1, 2, HC, pb) : 1 ∧ 2

Patrol Boat MVar

GC, HC

LIO Example

PI 1

: (1, 2, HC, pb) : 1 ∧ 2

: (1, 2, GC, pb) : 1 ∧ 2

PI 2

1 : (1, 2, GC, pb) : 1 ∧ 2

1 : (1, 2, HC, pb) : 1 ∧ 2

Yields Hit

: (1, 2, HC, pb) : 1 ∧ 2

: (1, 2, GC, pb) : 1 ∧ 2

Patrol Boat
MVar

GC, HC

LIO Example

PI 1		PI 2
: (1, 2, HC, pb) : 1 ∧ 2		1 : (1, 2, GC, pb) : 1 ∧ 2
: (1, 2, GC, pb) : 1 ∧ 2		1 : (1, 2, HC, pb) : 1 ∧ 2
	Yields Hit	: (1, 2, HC, pb) : 1 ∧ 2
	Yields Sank PatrolBoat	: (1, 2, GC, pb) : 1 ∧ 2

Patrol Boat MVar	GC, HC

LIO Example

PI 1

: (1, 2, HC, pb) : 1 ∧ 2

: (1, 2, GC, pb) : 1 ∧ 2

PI 2

1 : (1, 2, GC, pb) : 1 ∧ 2

1 : (1, 2, HC, pb) : 1 ∧ 2

: (1, 2, HC, pb) : 1 ∧ 2

: (1, 2, GC, pb) : 1 ∧ 2

Still Yields Hit

Yields Sank PatrolBoat

Patrol Boat
MVar

GC, HC

Concurrent ML

- Concurrent ML is a library for Standard ML (we use the Standard ML of New Jersey implementation)
- It has no special security features
- But the combination of its abstract types (provided by its rich module system) and mutable references can be used to program access control

CML + AC Battleship

- PIs exchange — using trusted code — immutable, abstract locked boards, whose cells can be unlocked using unforgeable keys held by originating player:

```
type key (* key *)
type ck  (* counted key *)
val labelKey : key * int -> ck
type lb  (* locked board *)
datatype lsr =
           Invalid      (* invalid counted key *)
         | Repeat       (* illegal repetition *)
         | Miss         (* missed a ship *)
         | Hit          (* hit an unspecified ship *)
         | Sank of ship (* sank the given ship *)
val lockedShoot : lb * pos * ck -> lb * lsr
```

CML + AC Example

PI 1

PI 2

lb_1

CML + AC Example

PI 1

PI 2

Ib_1

HC

CML + AC Example

PI 1

HC

PI 2

Ib_1 HC

CML + AC Example

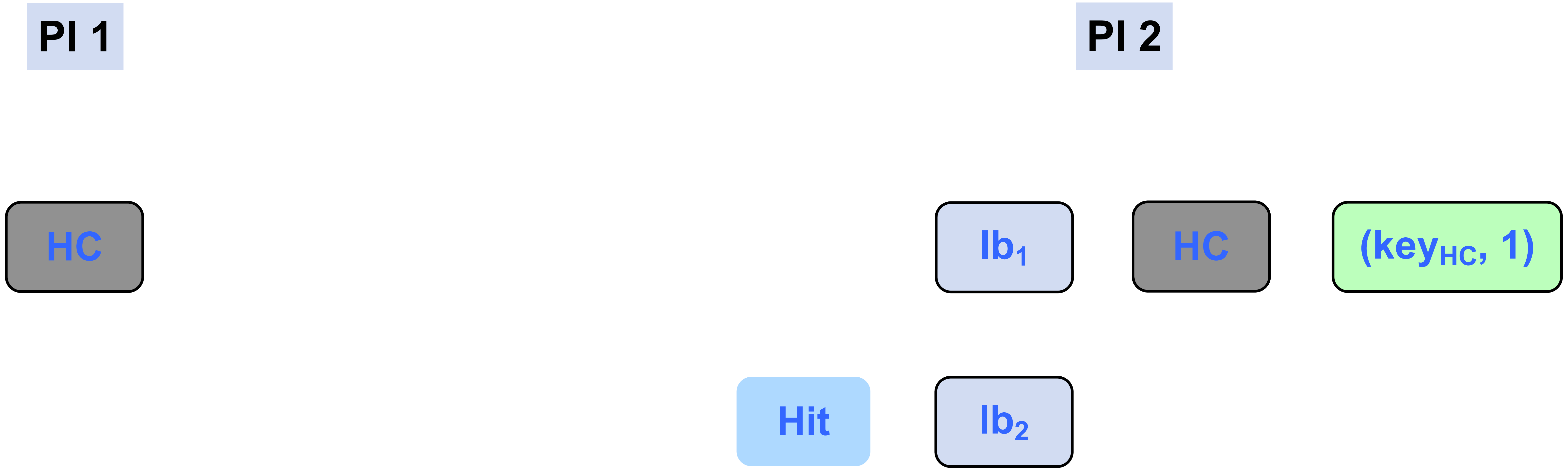
PI 1

HC

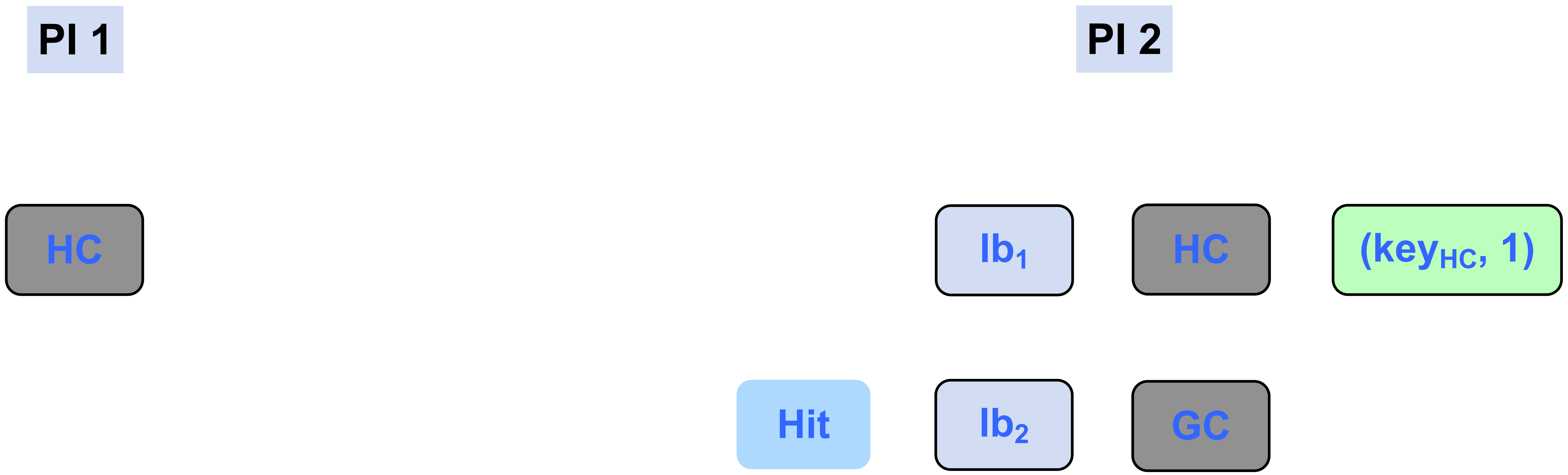
PI 2

lb_1 HC $(key_{HC}, 1)$

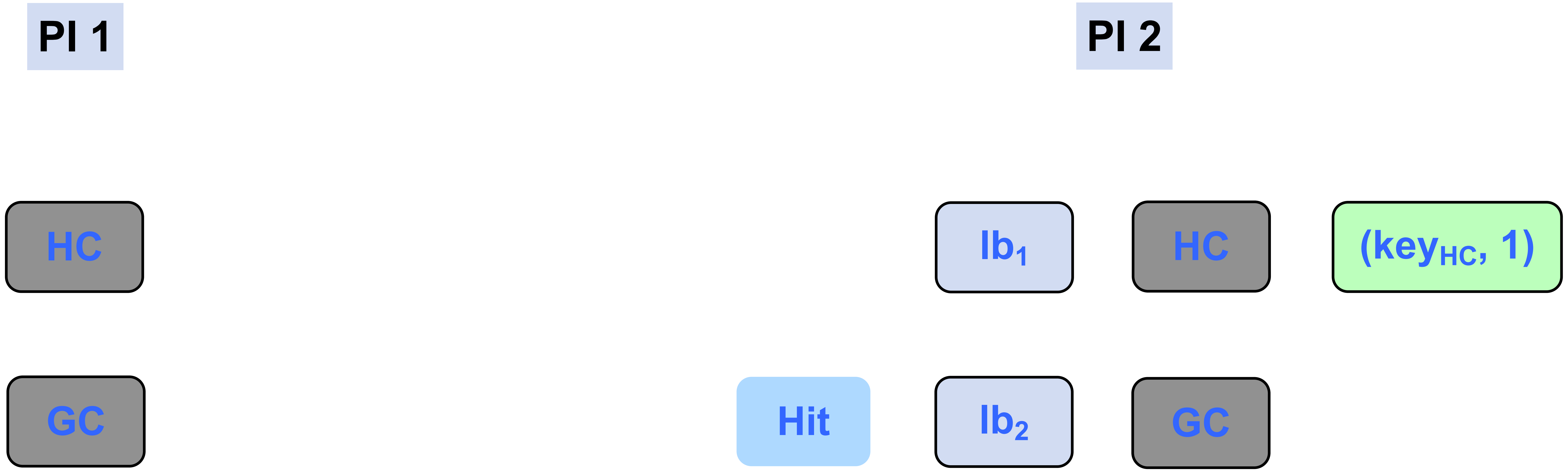
CML + AC Example



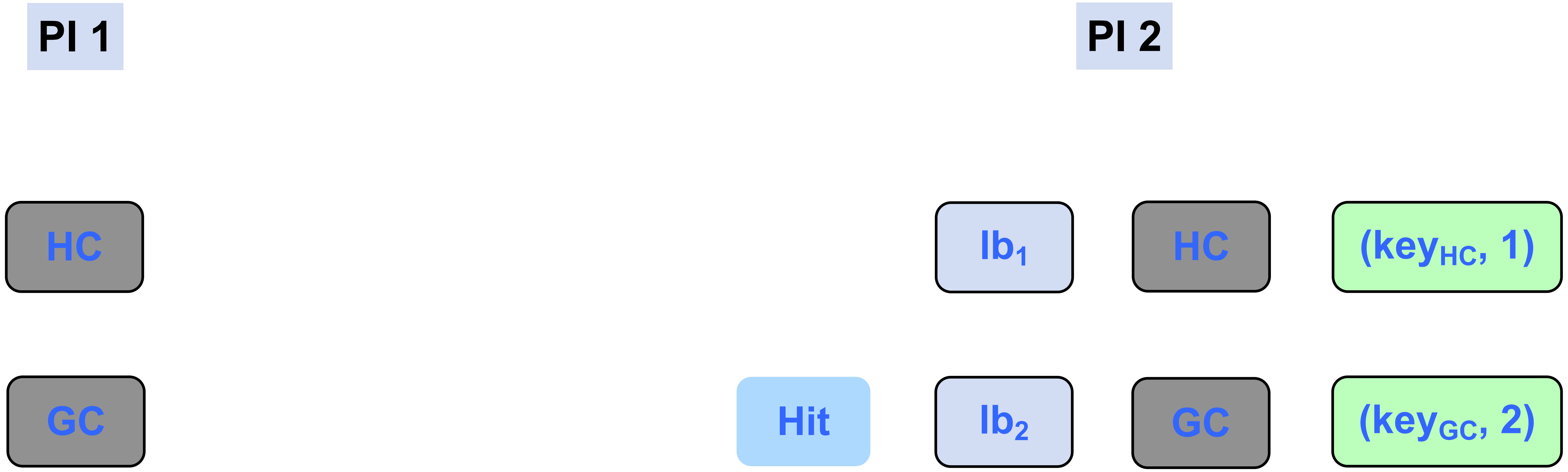
CML + AC Example



CML + AC Example



CML + AC Example

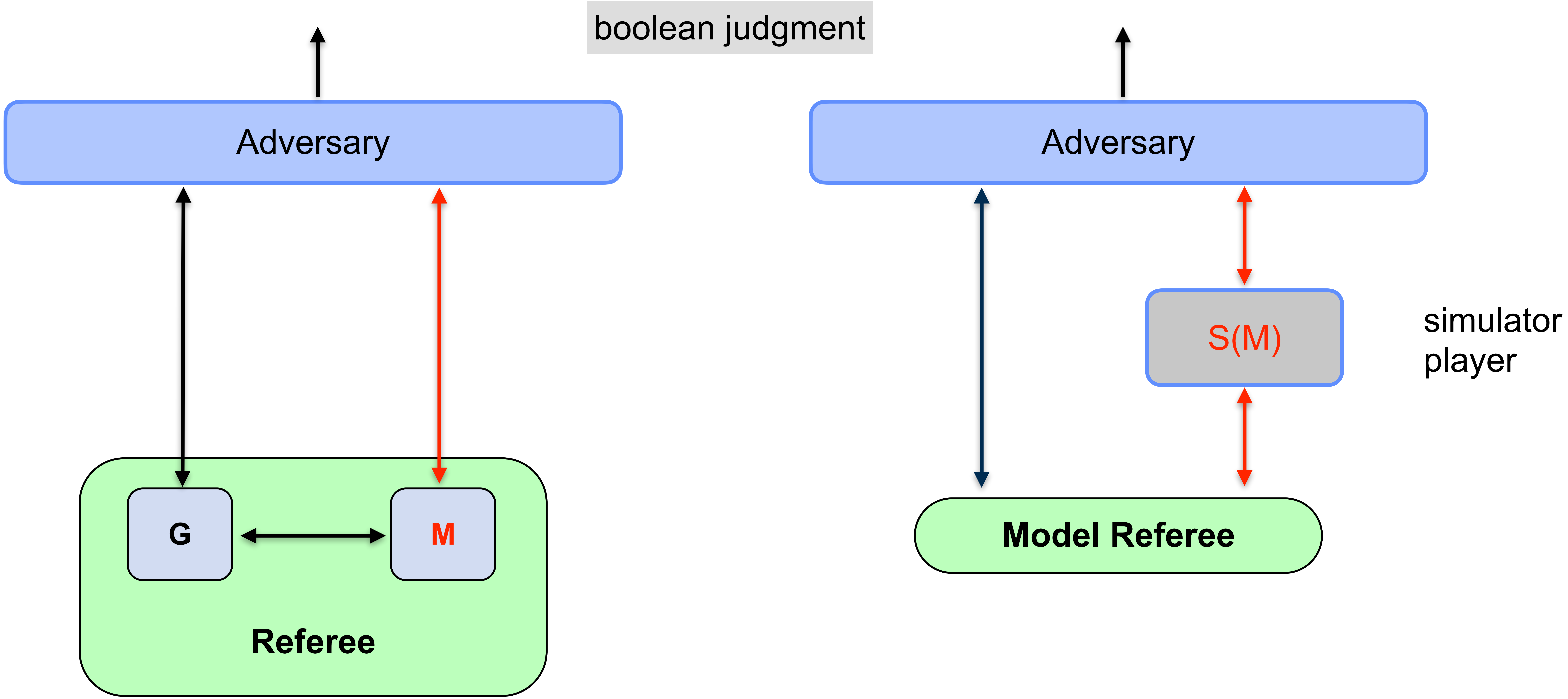


CML + AC Example

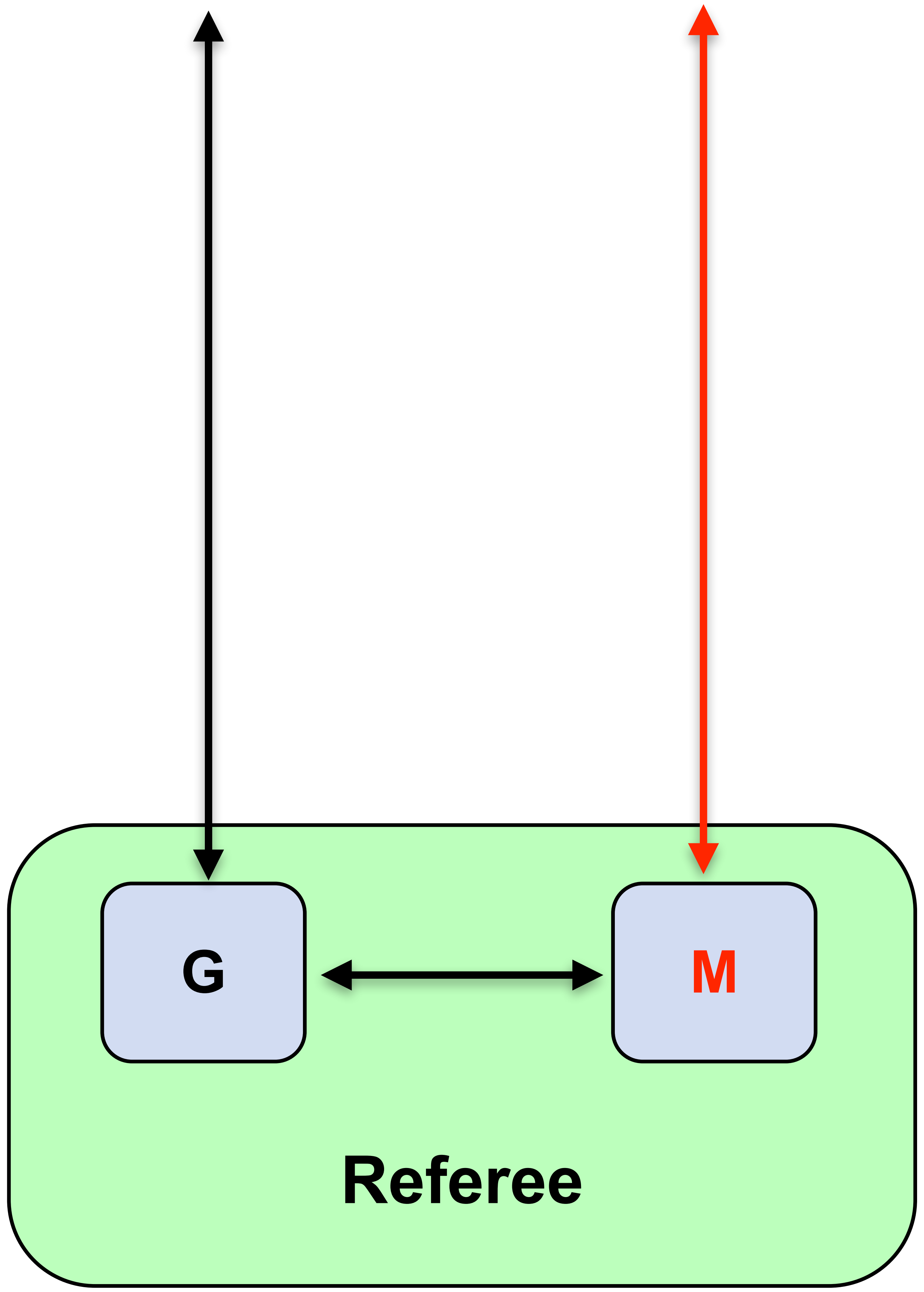
PI 1			PI 2		
HC		lb_1	HC	(key_{HC}, 1)
GC	Hit	lb_2	GC	(key_{GC}, 2)
	Sank PatrolBoat	lb_3		

A counted key is only applicable to a single locked board, and can't be deconstructed

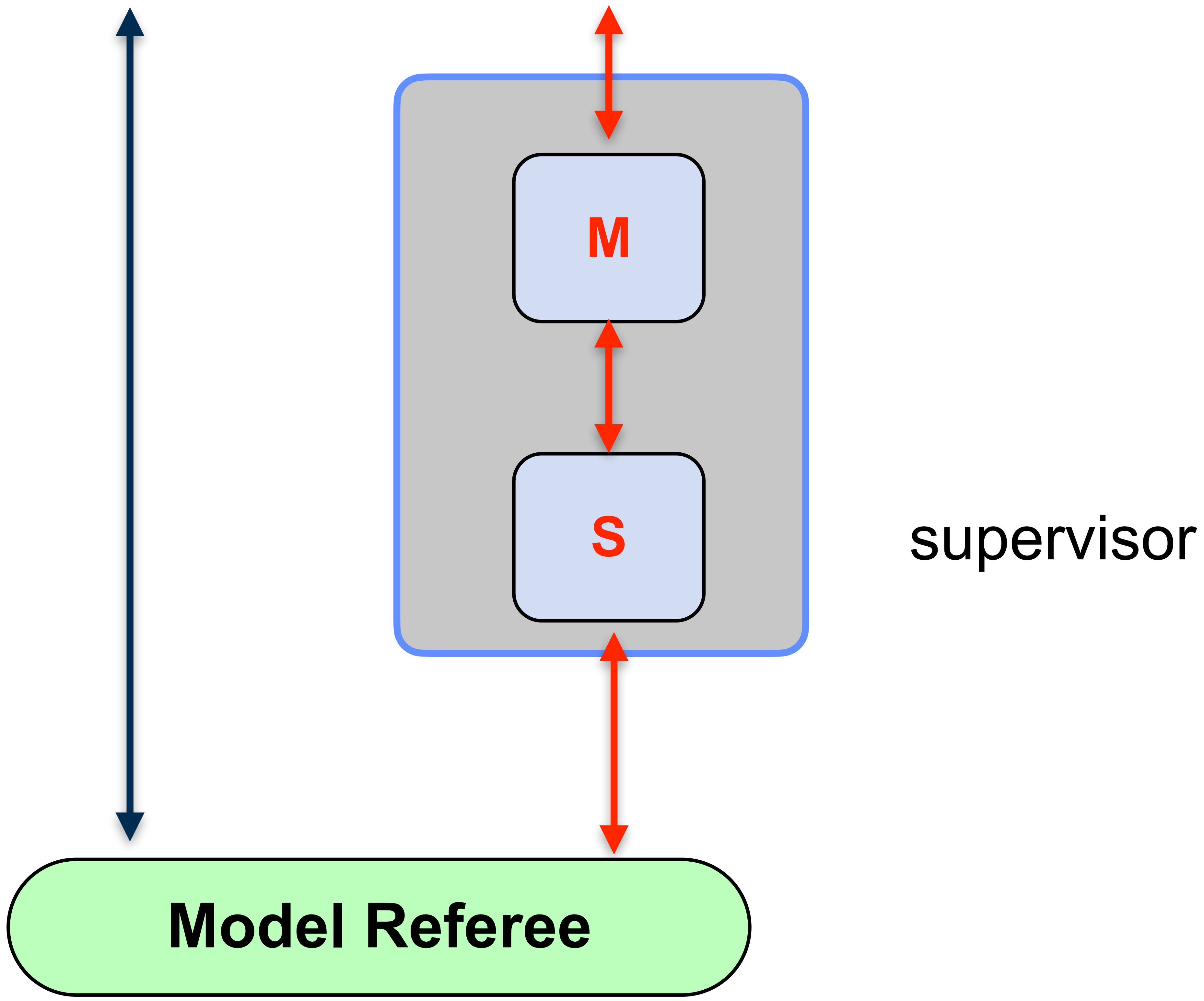
Construction of Simulator Player for CML + AC



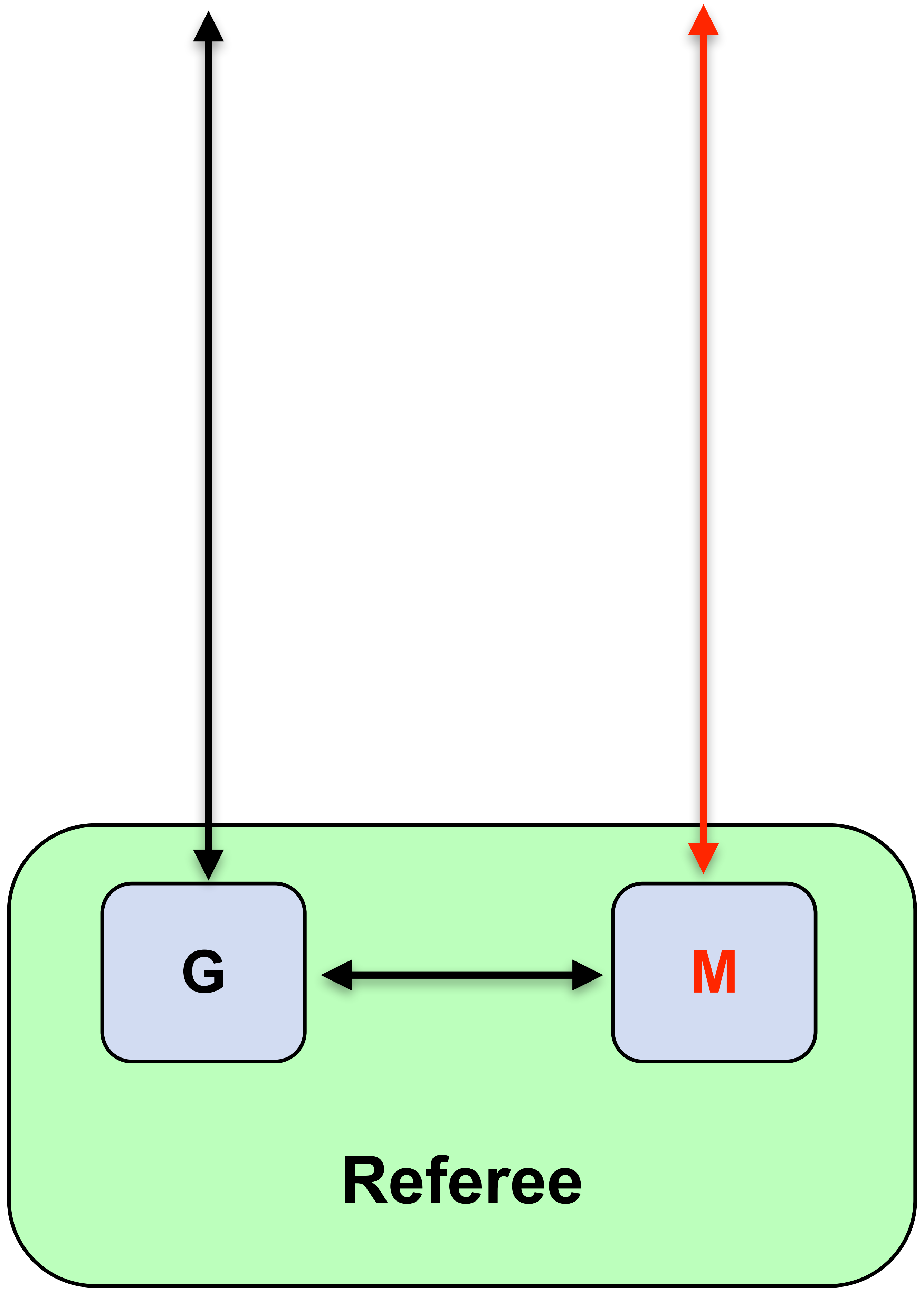
Construction of Simulator Player for CML + AC

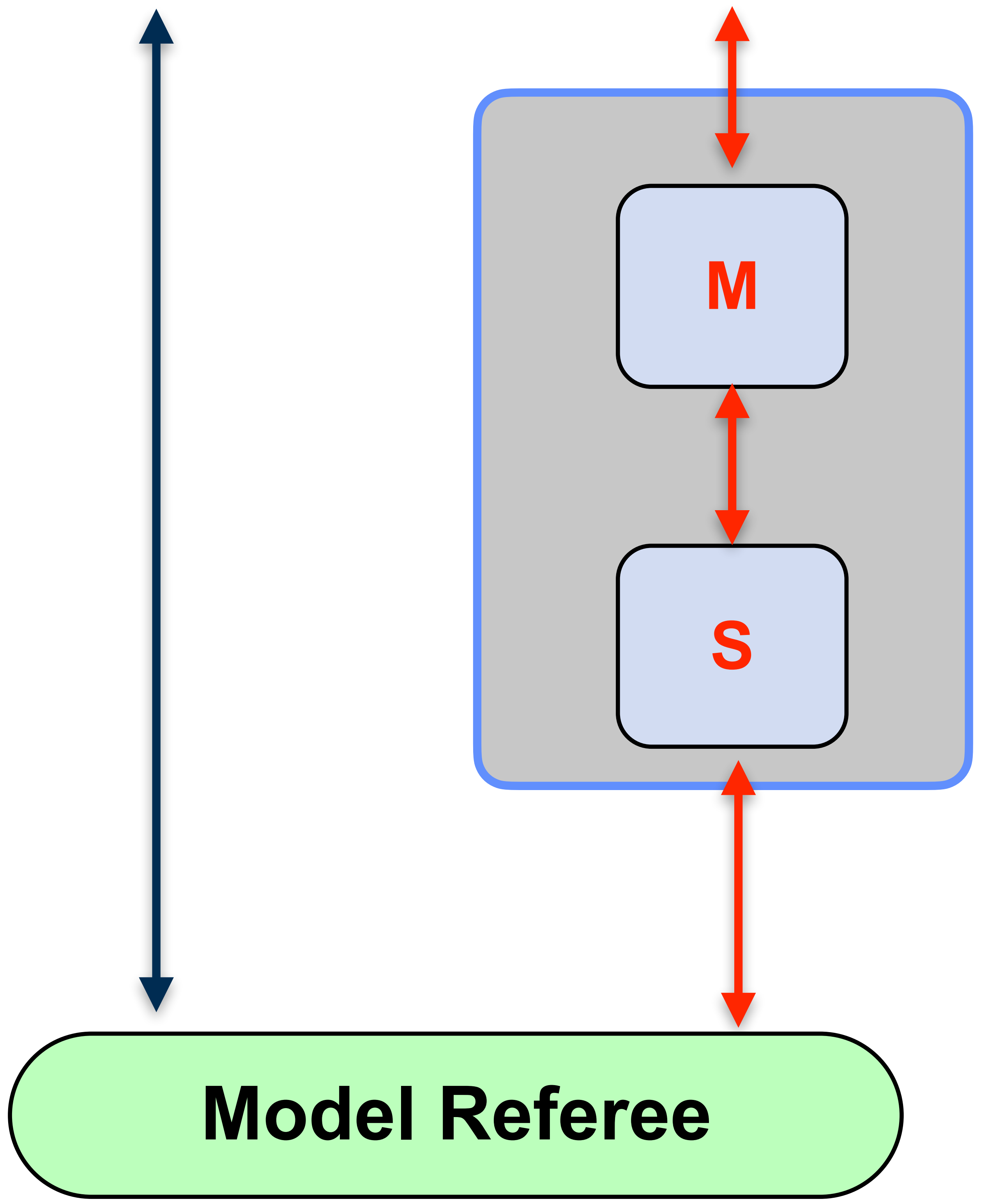


supervisor interacts with M using *reimplementation* of locked board abstract type

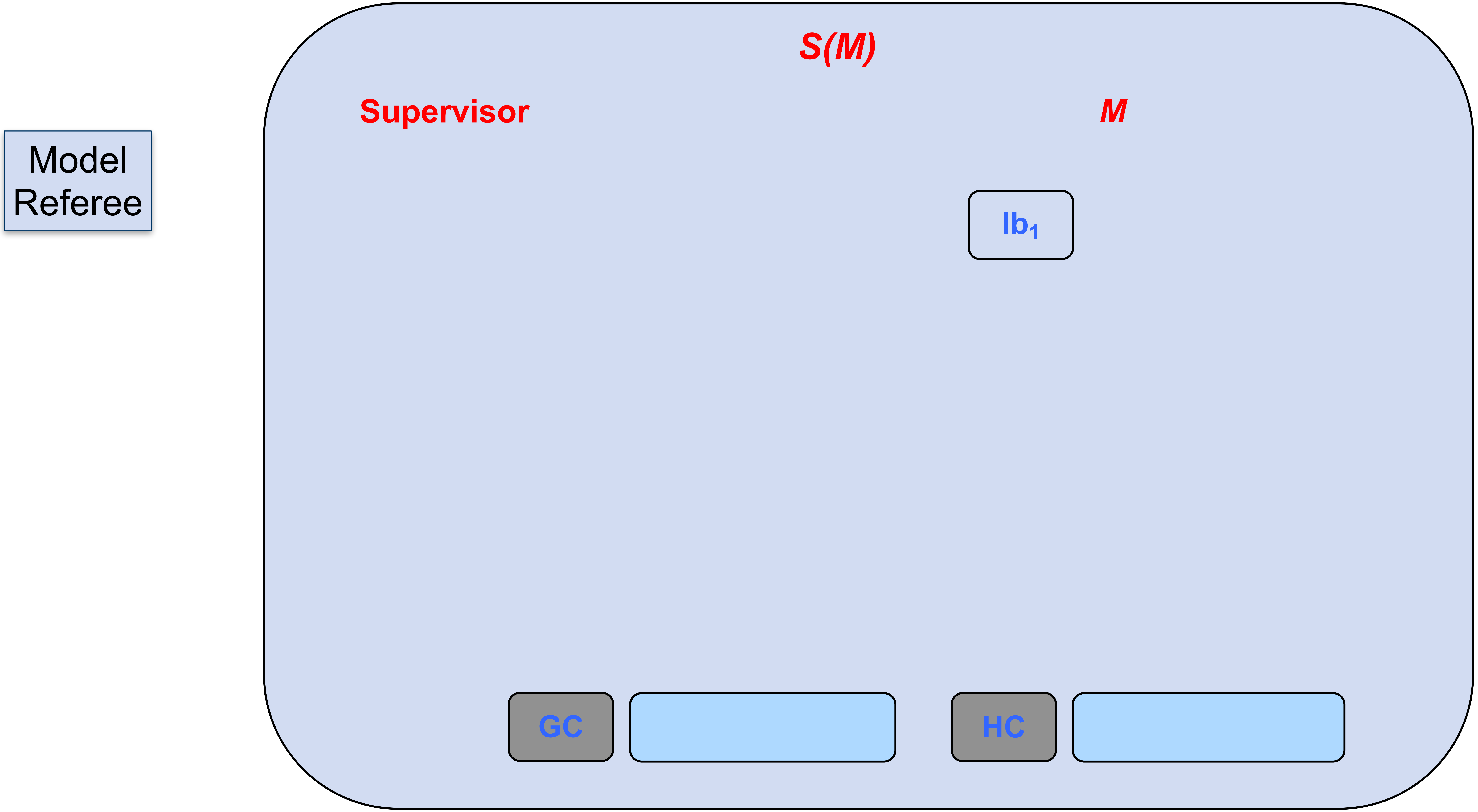


CML + AC: M Doesn't Learn More Than it Should

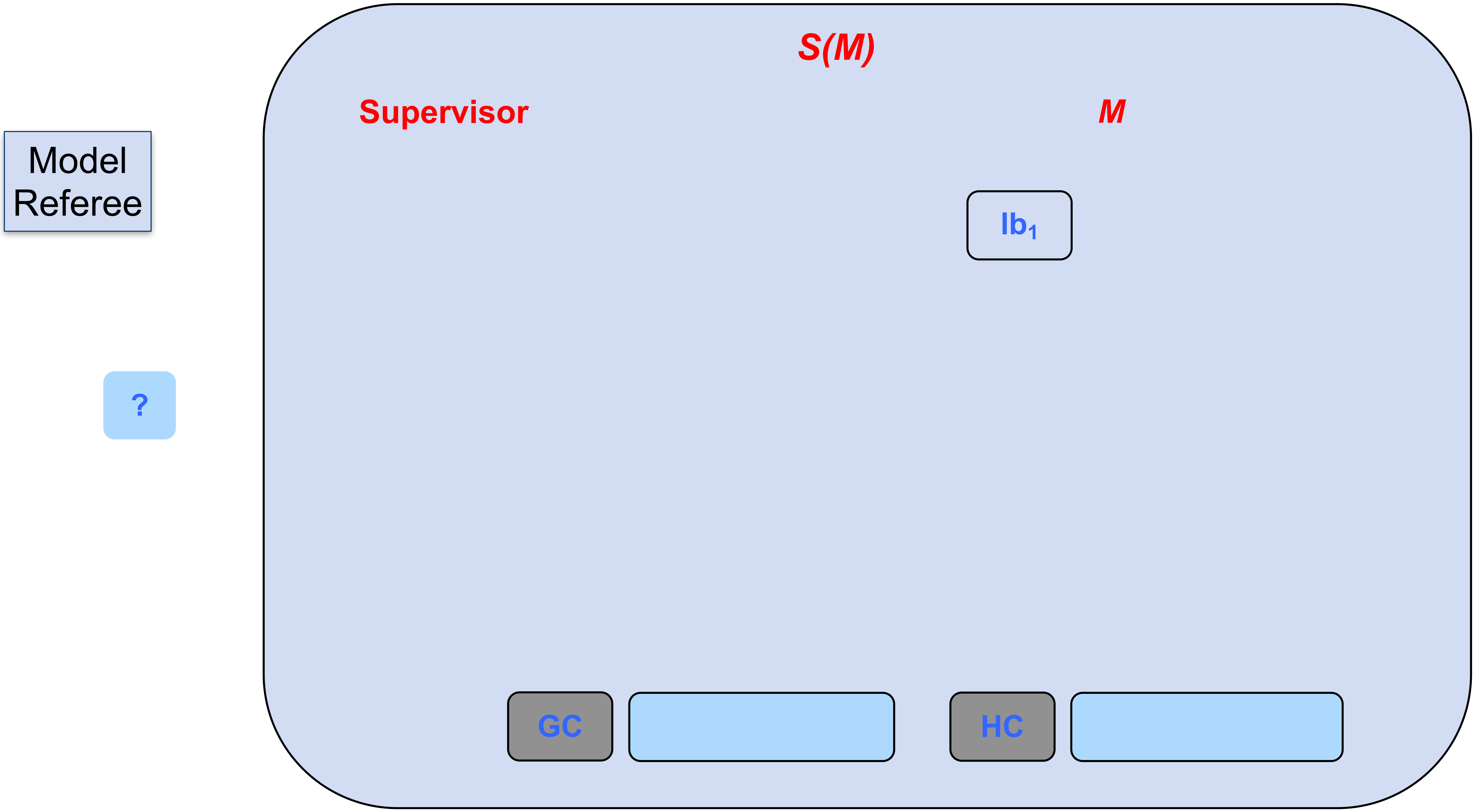




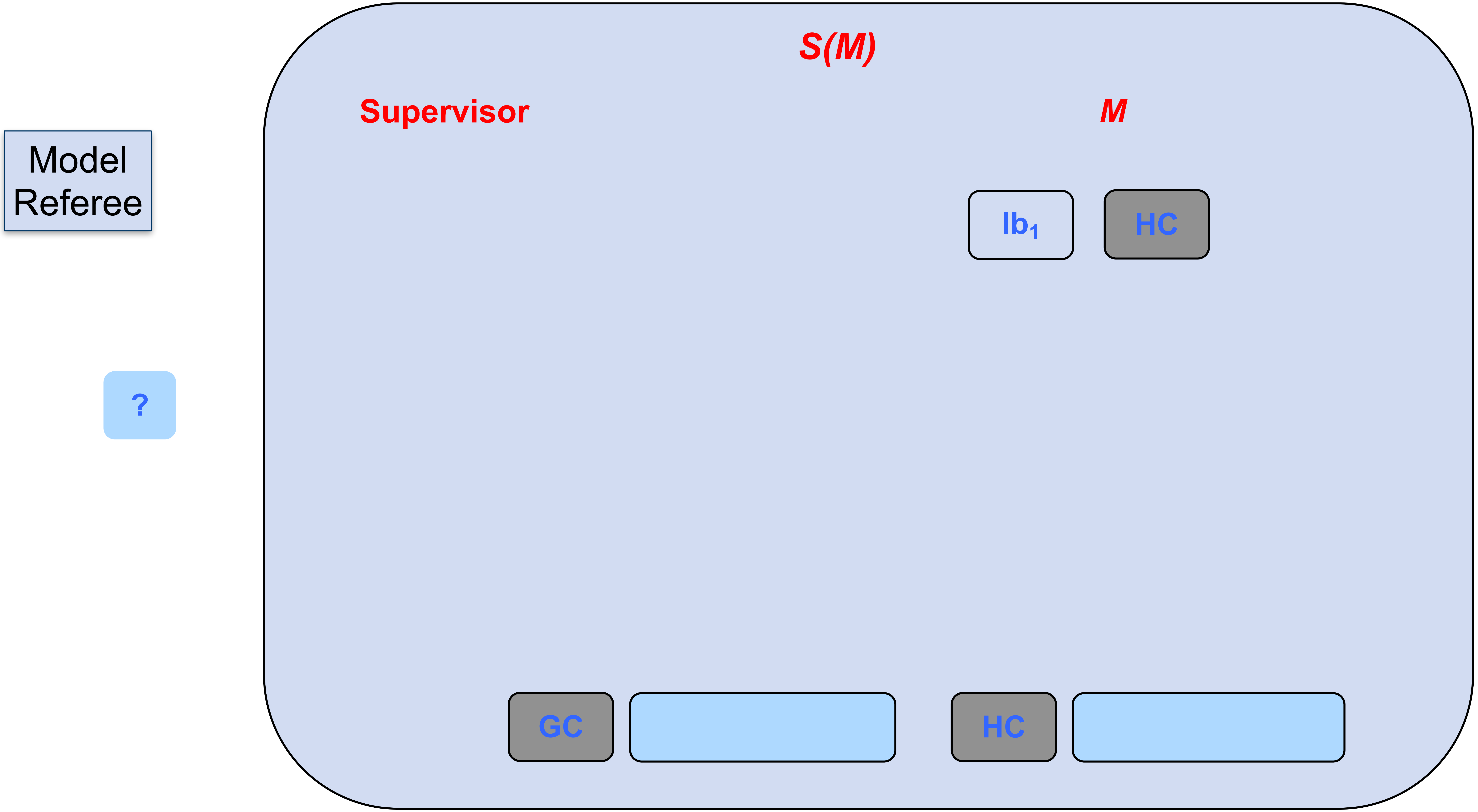
CML + AC Simulator Example



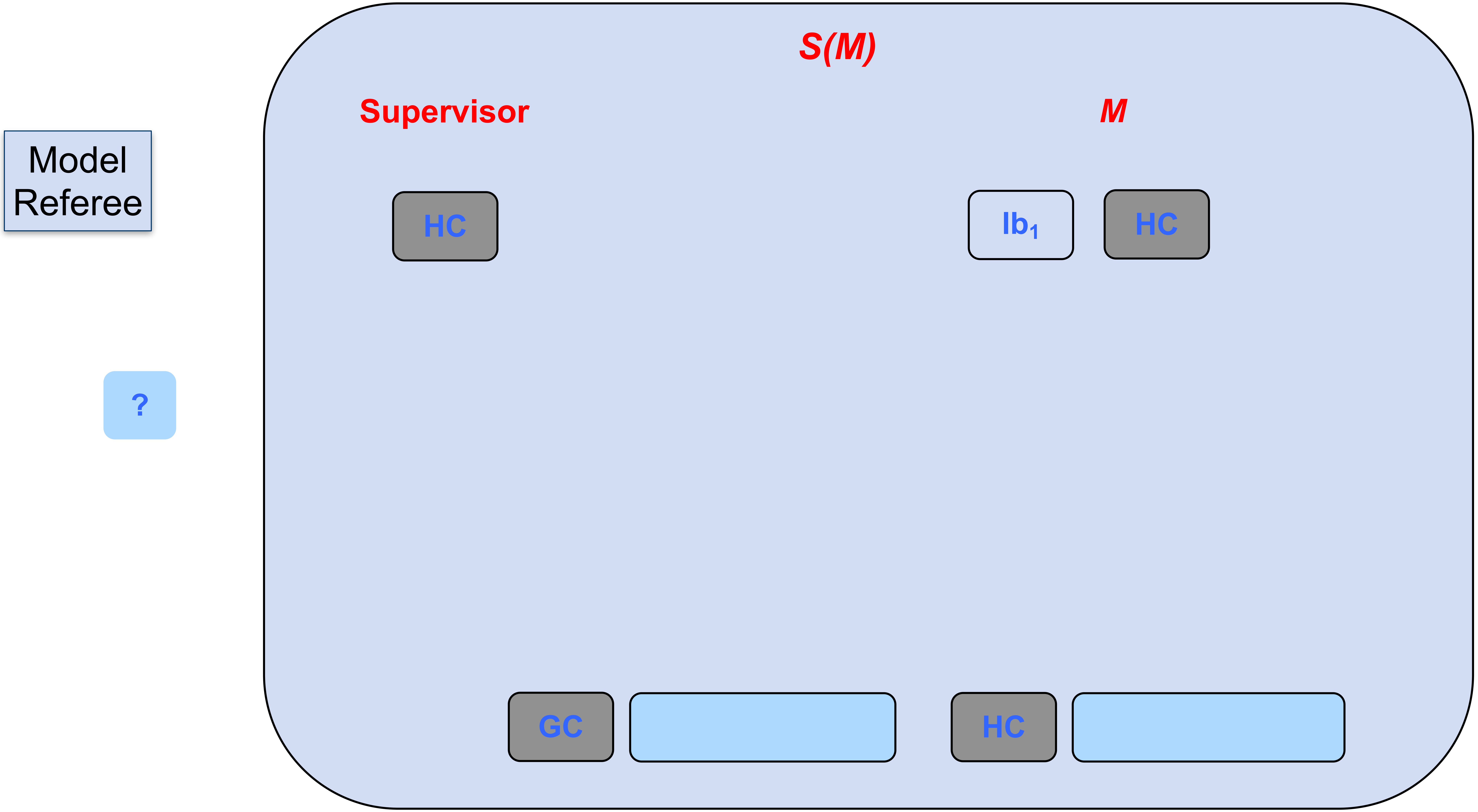
CML + AC Simulator Example



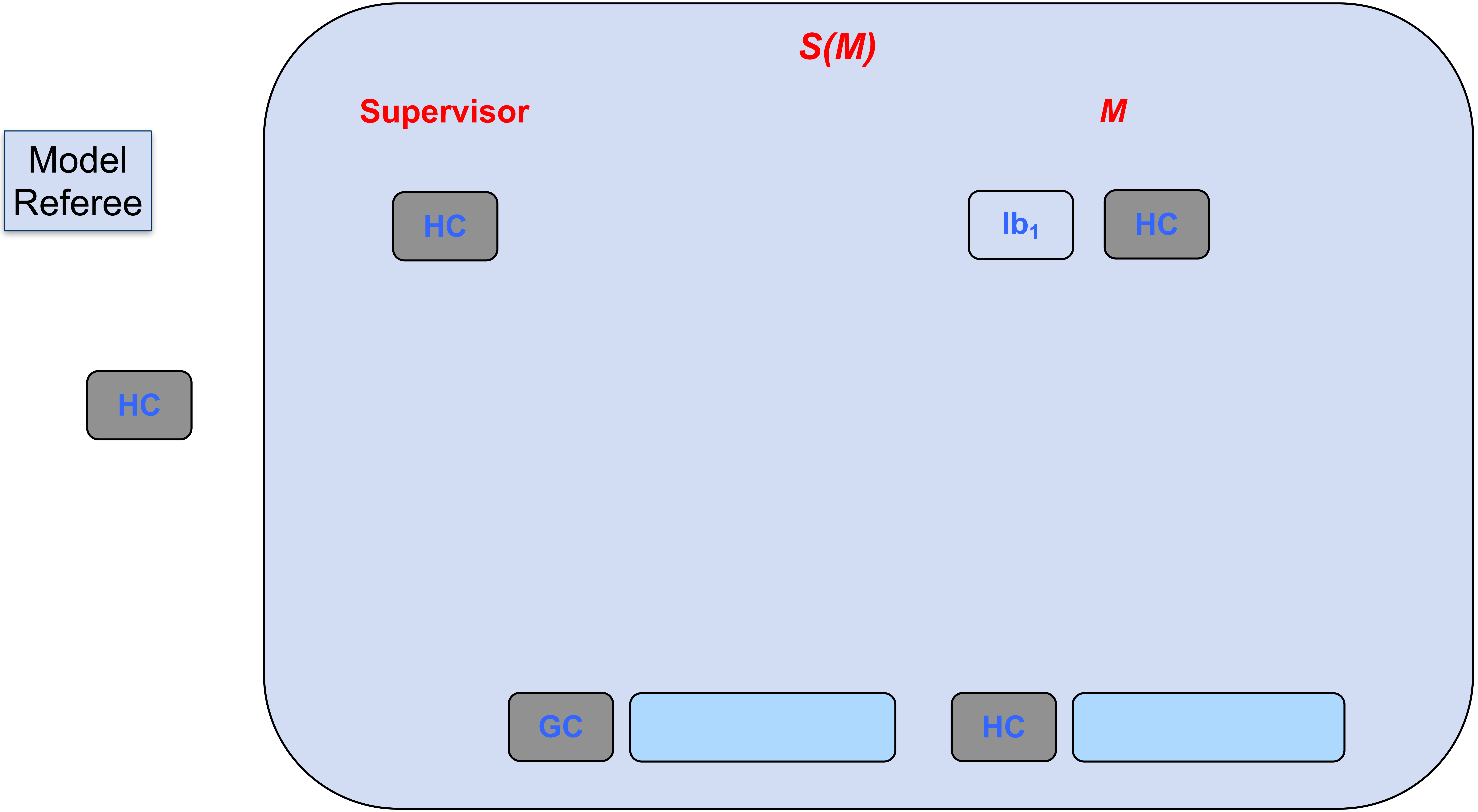
CML + AC Simulator Example



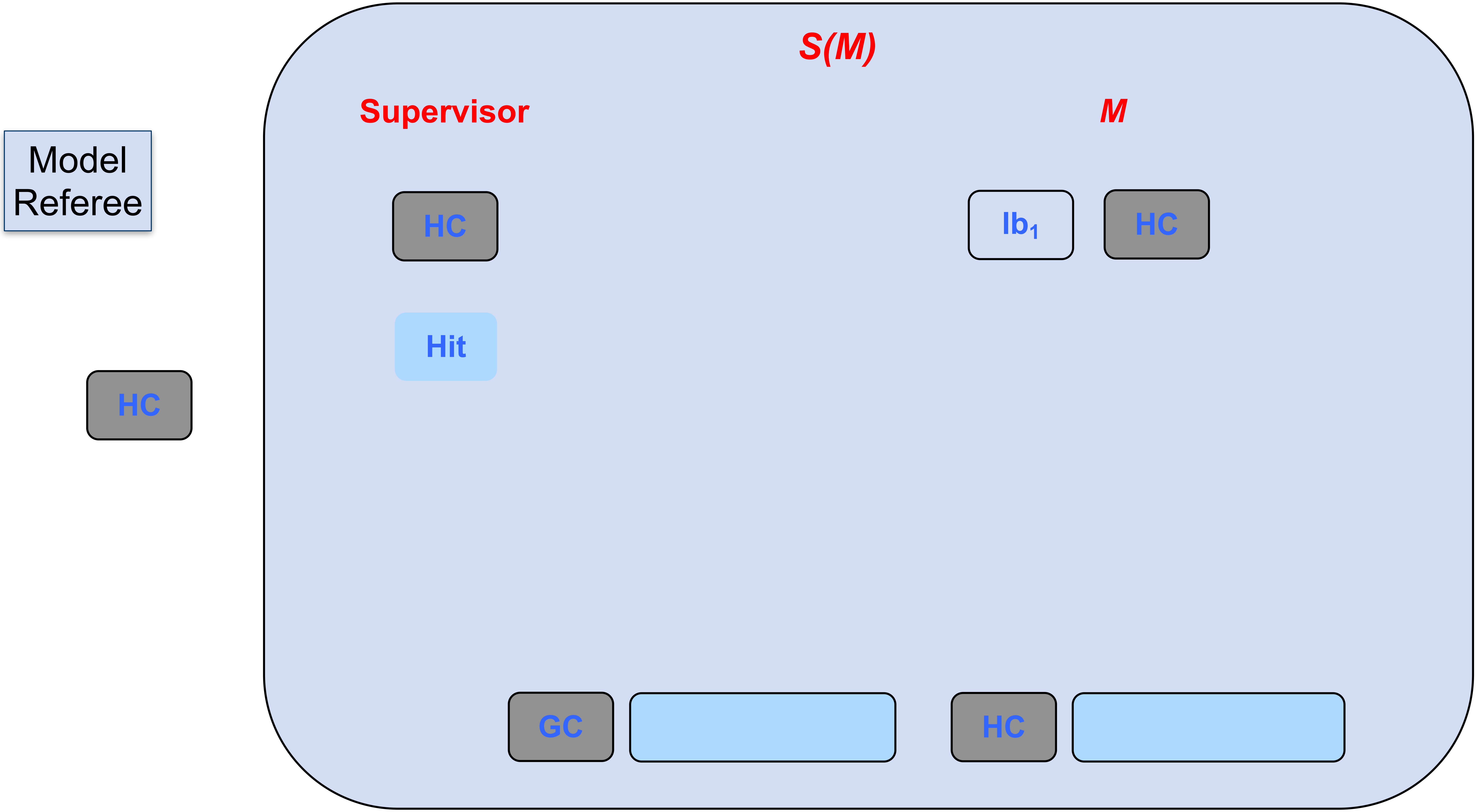
CML + AC Simulator Example



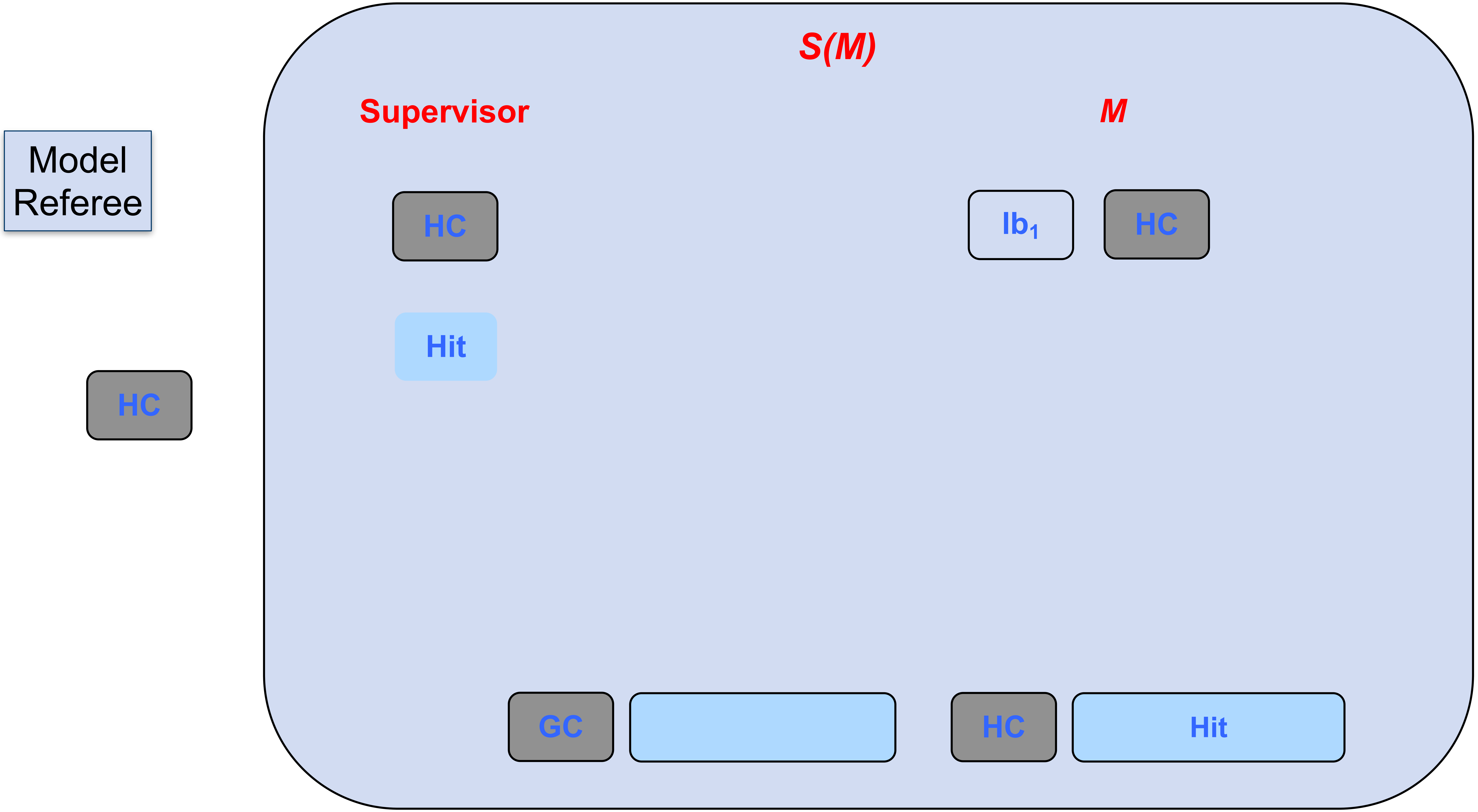
CML + AC Simulator Example



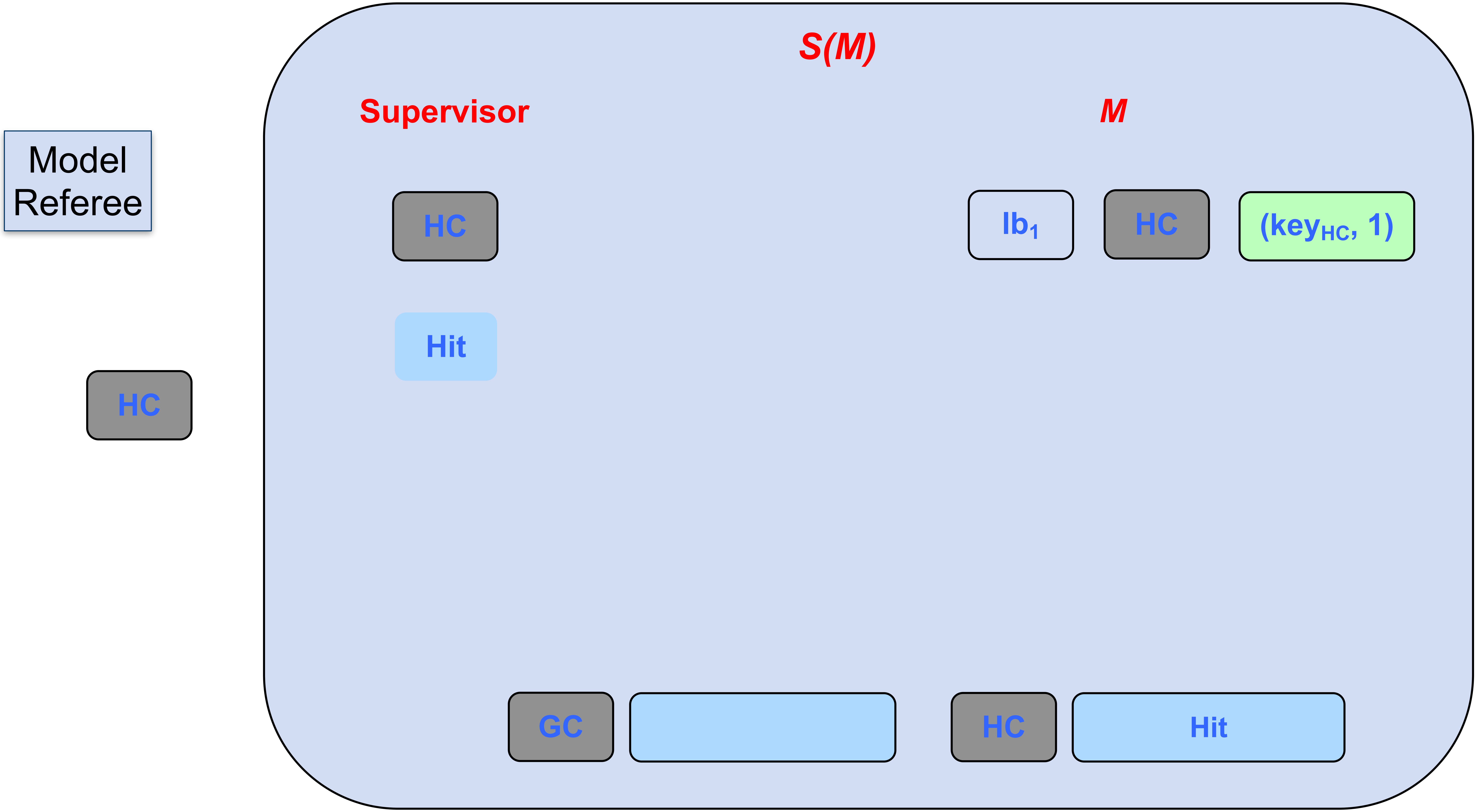
CML + AC Simulator Example



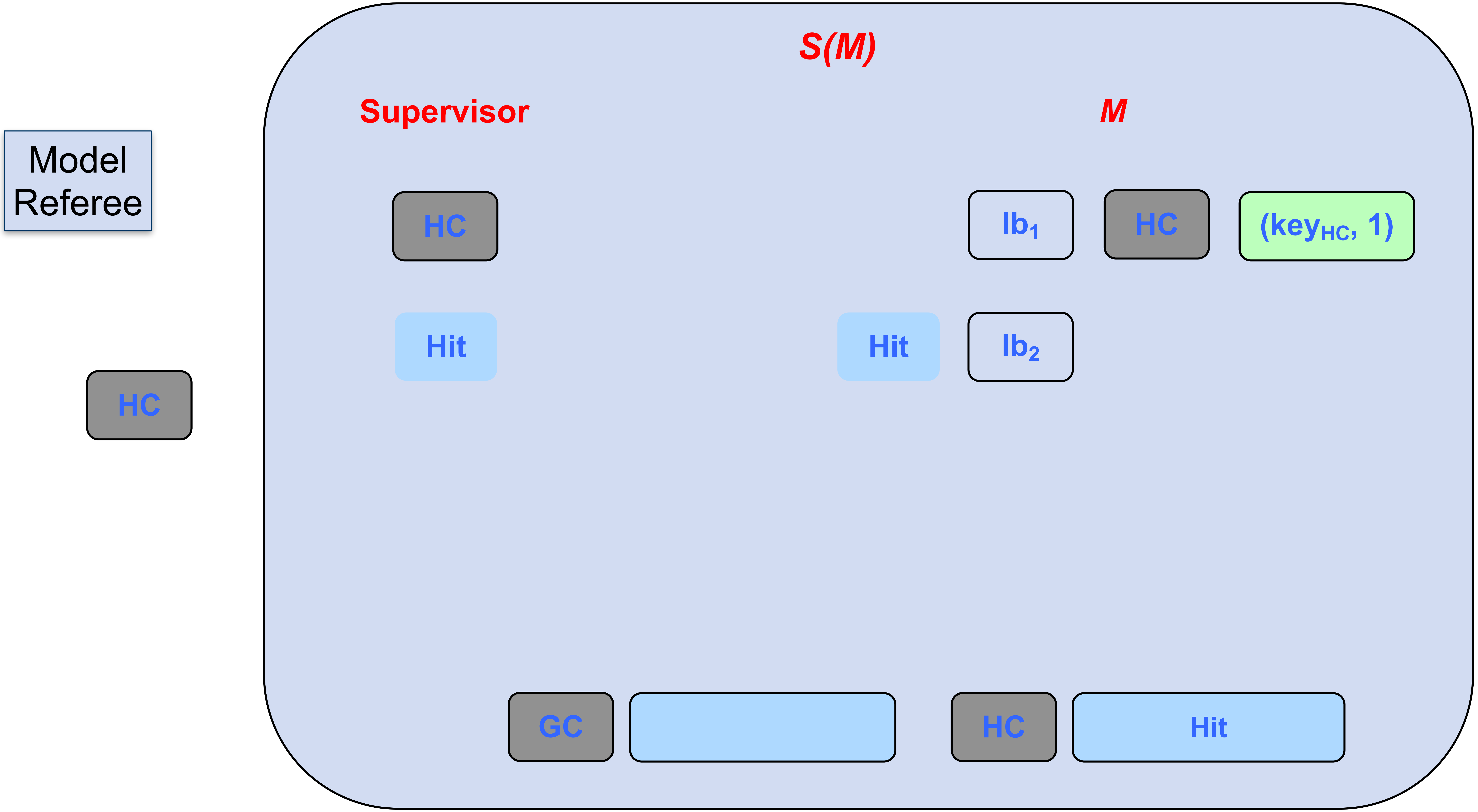
CML + AC Simulator Example



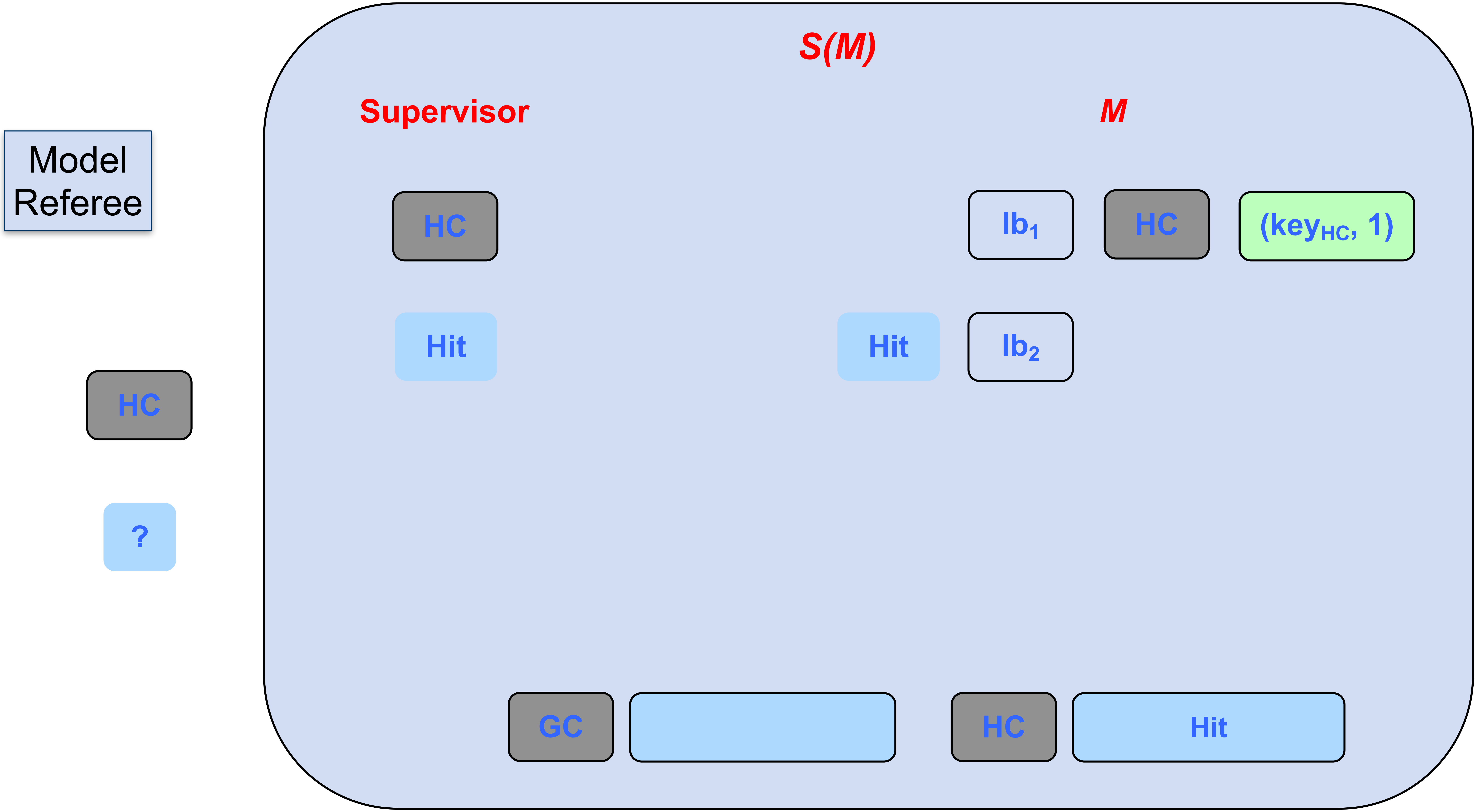
CML + AC Simulator Example



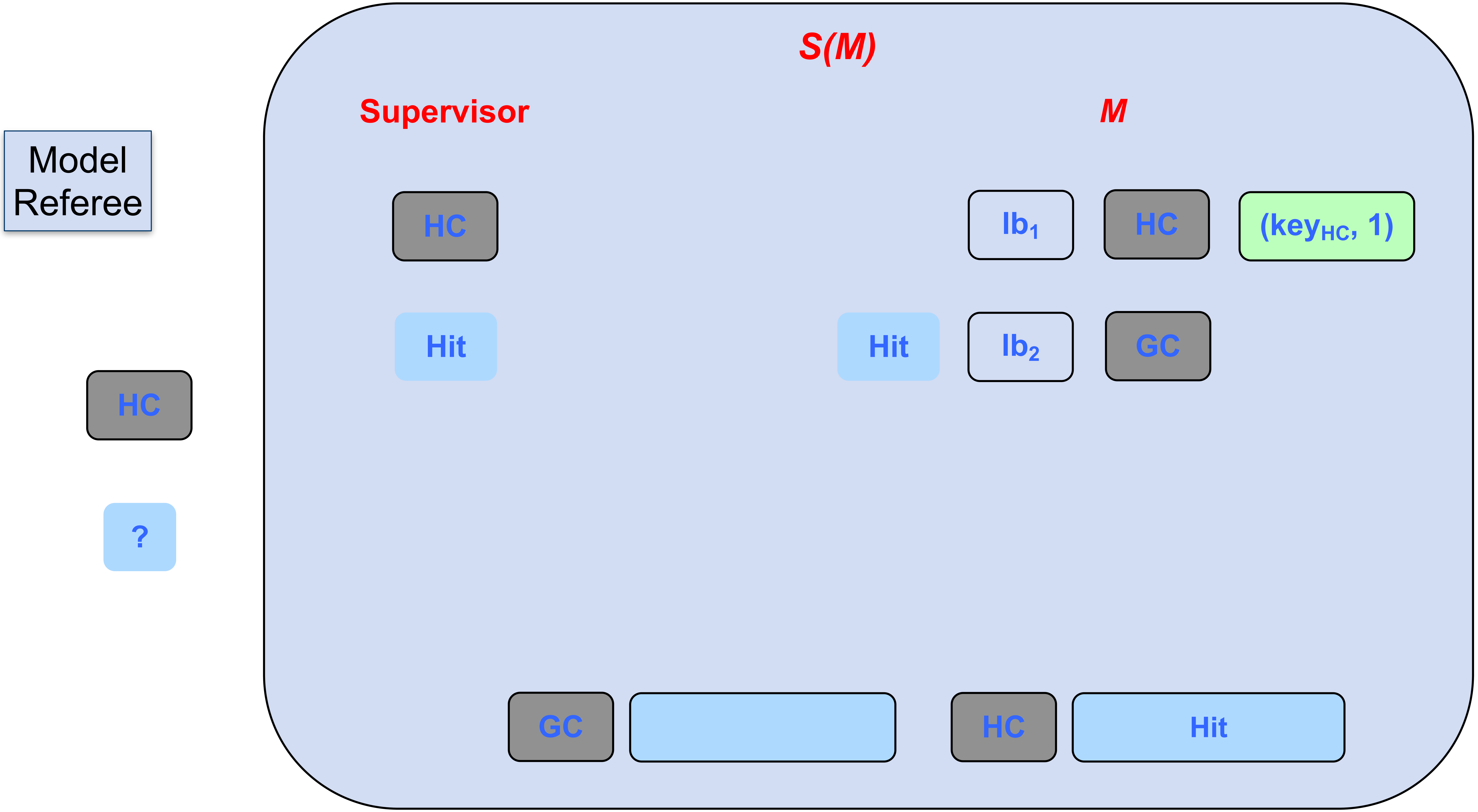
CML + AC Simulator Example



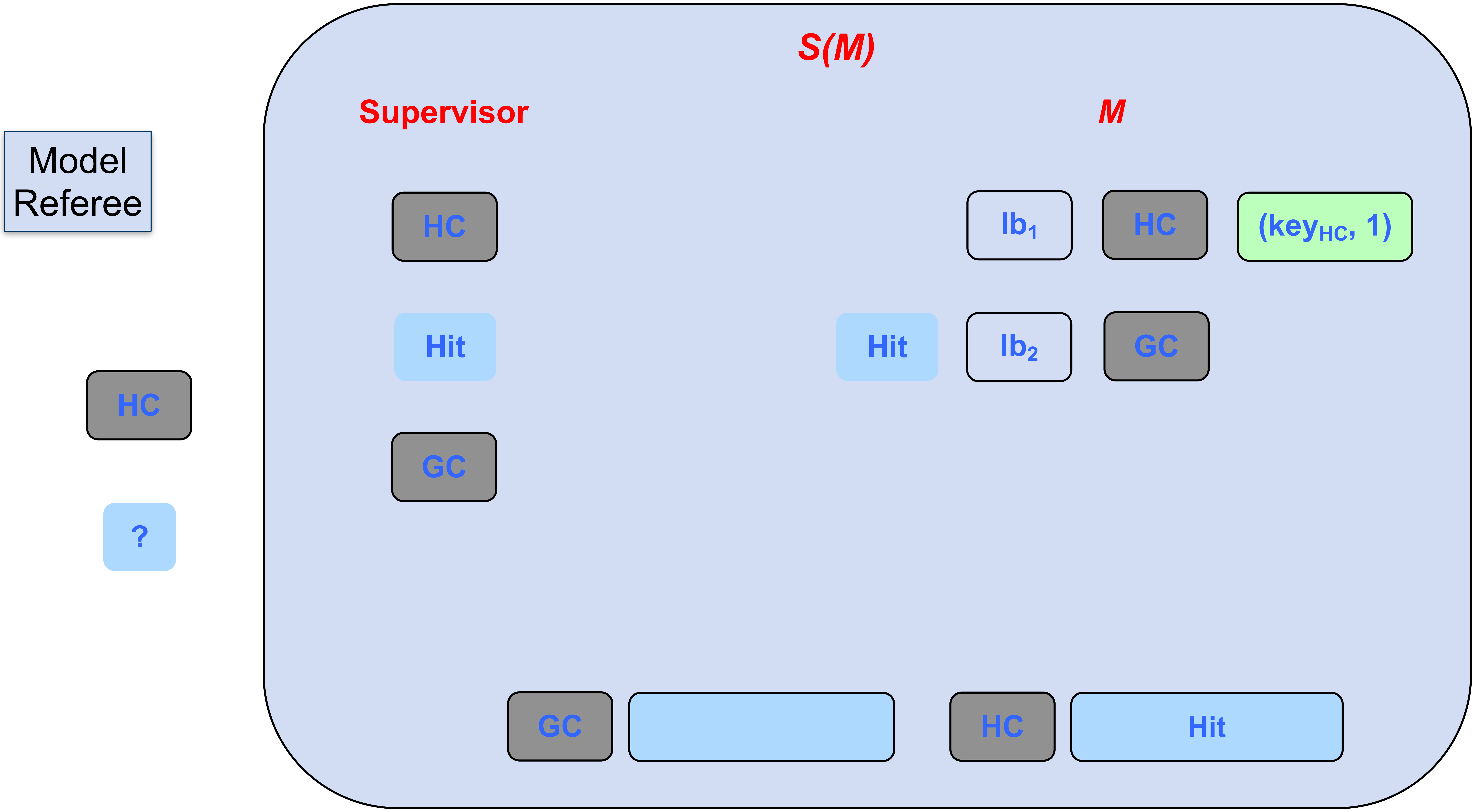
CML + AC Simulator Example



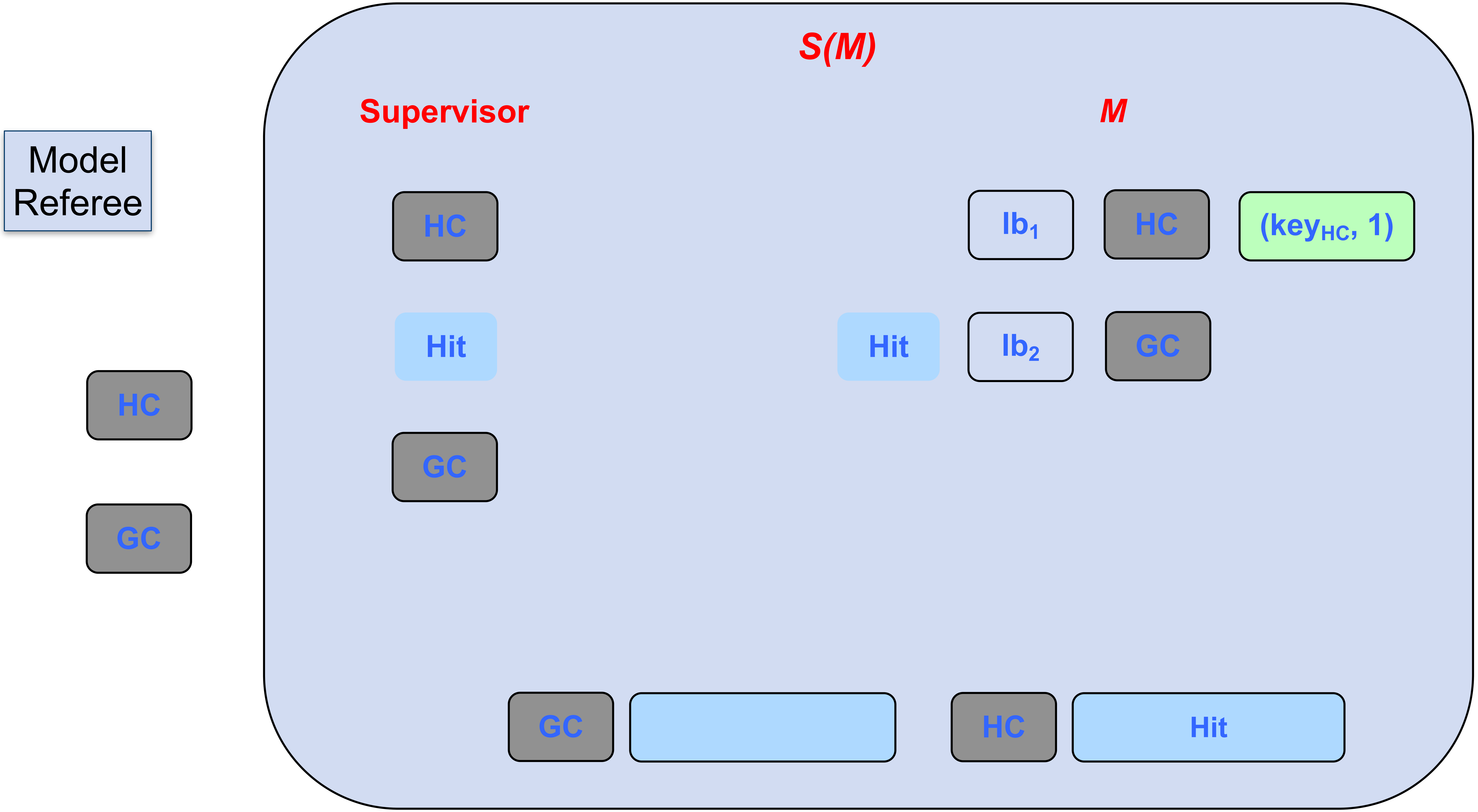
CML + AC Simulator Example



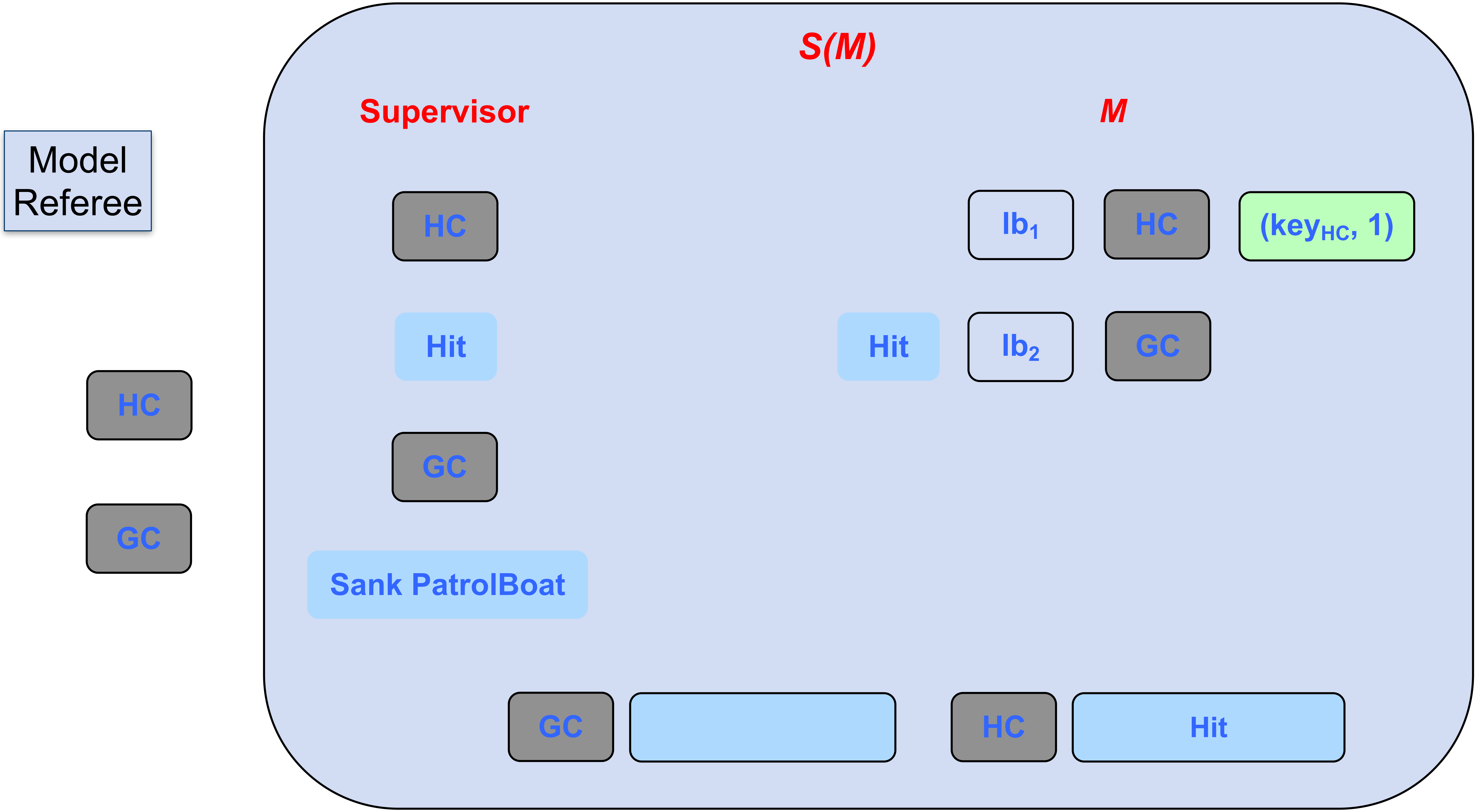
CML + AC Simulator Example



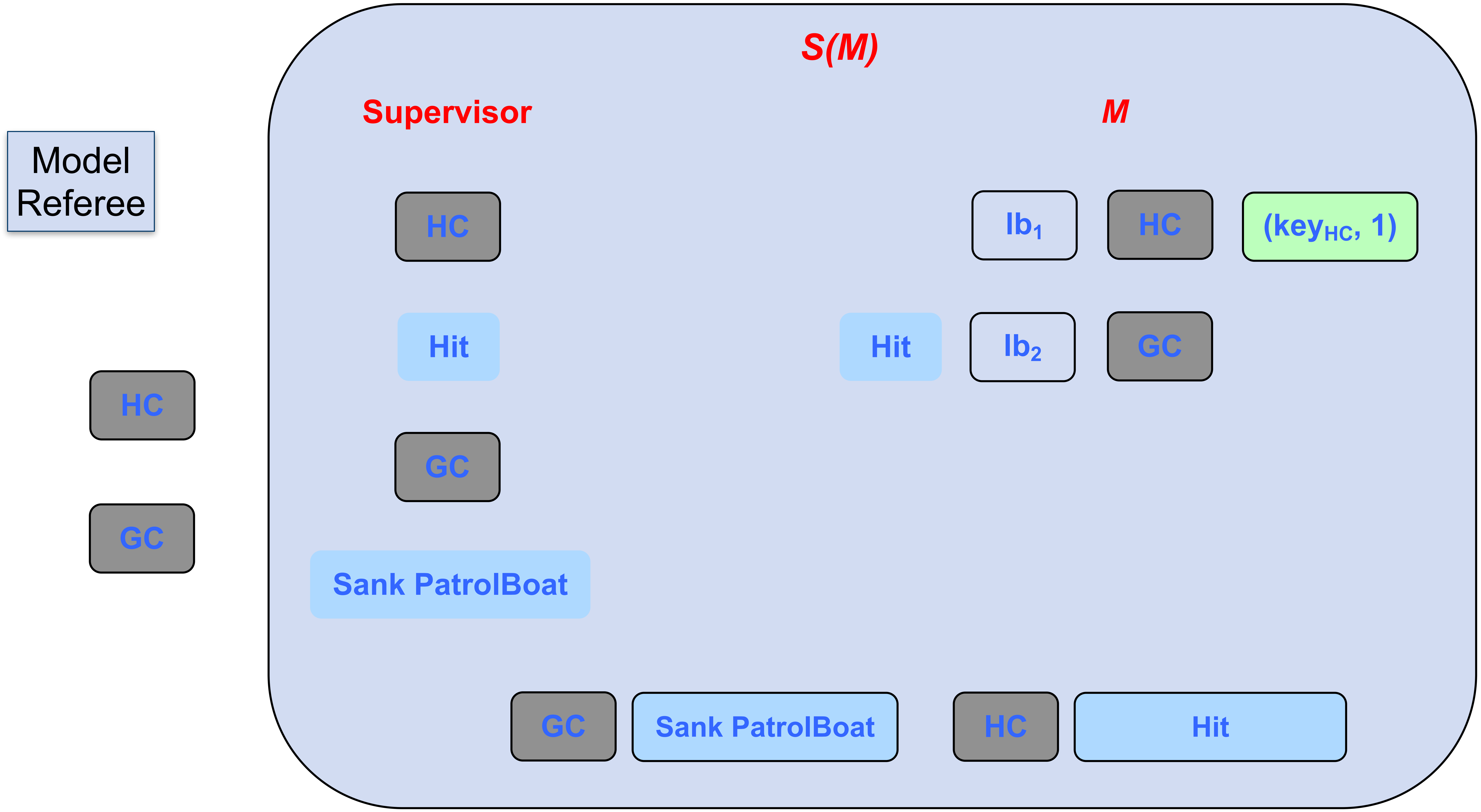
CML + AC Simulator Example



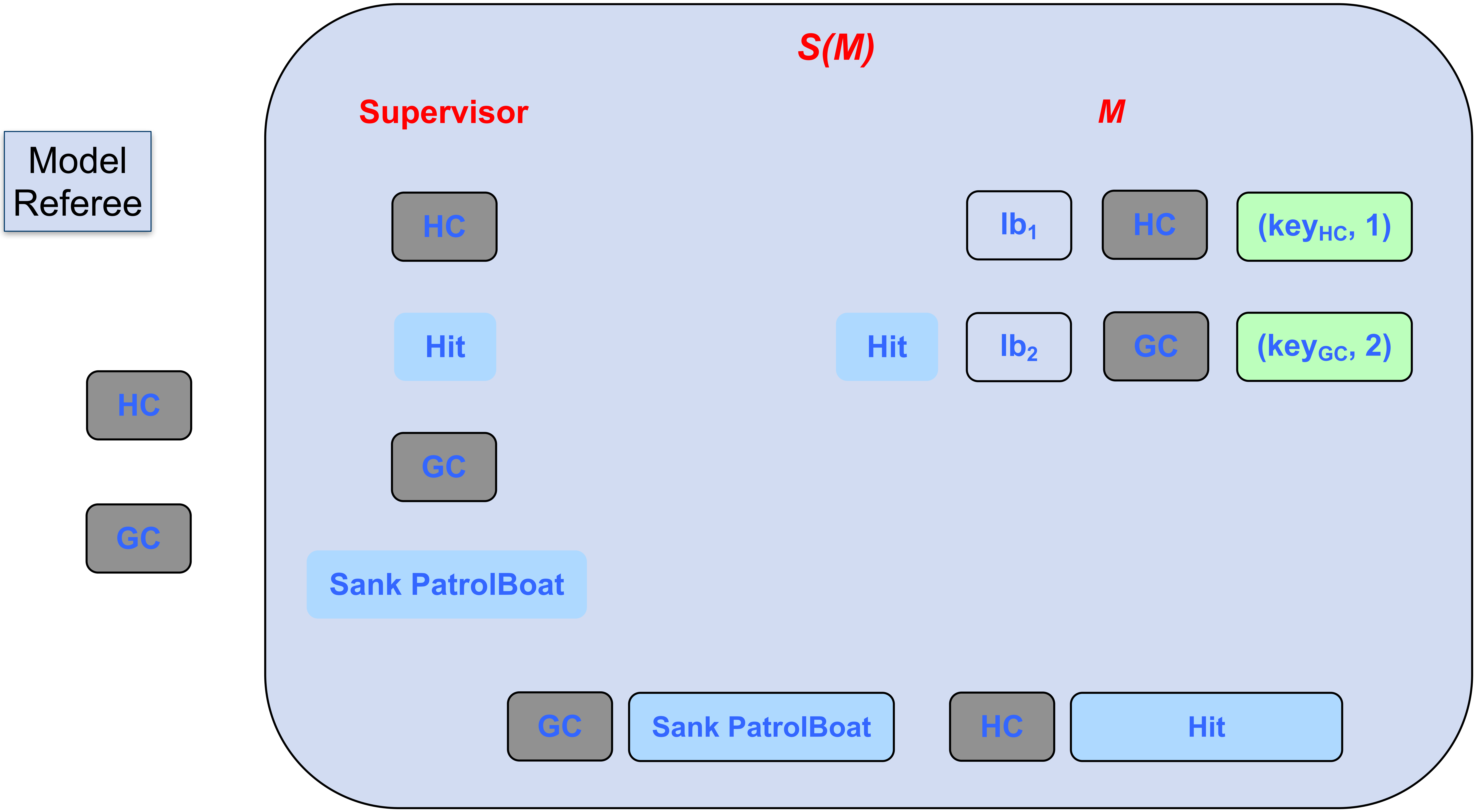
CML + AC Simulator Example



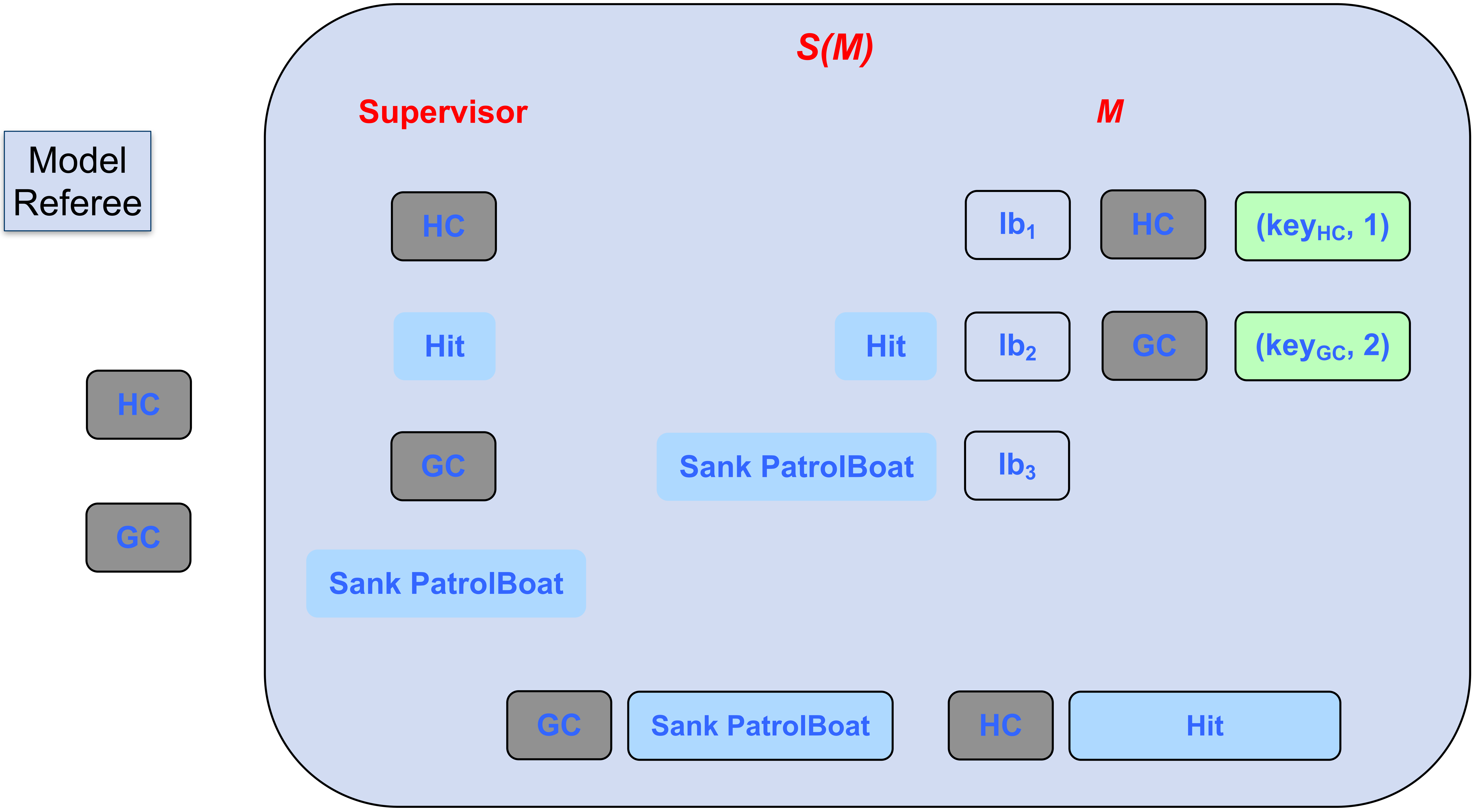
CML + AC Simulator Example



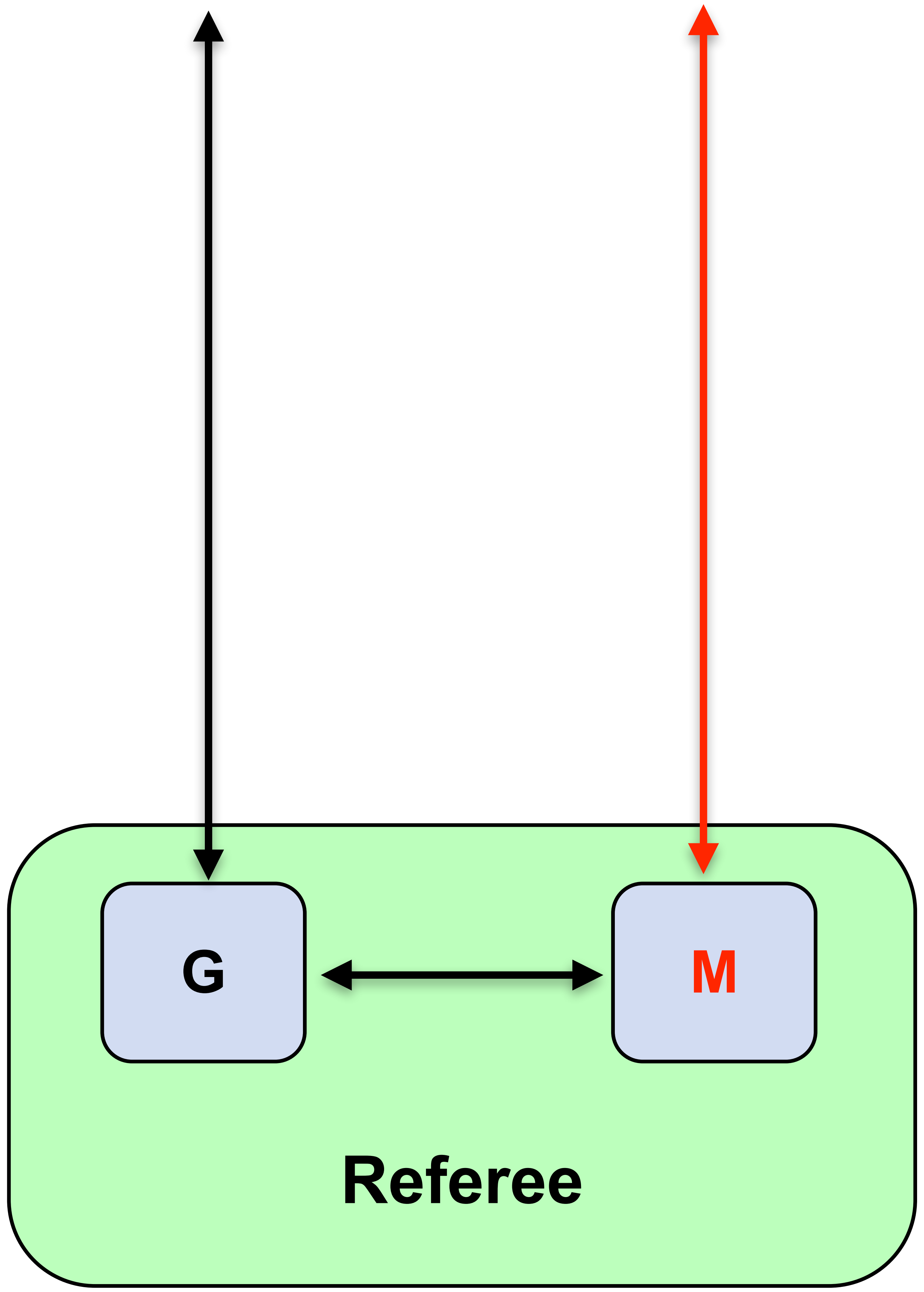
CML + AC Simulator Example



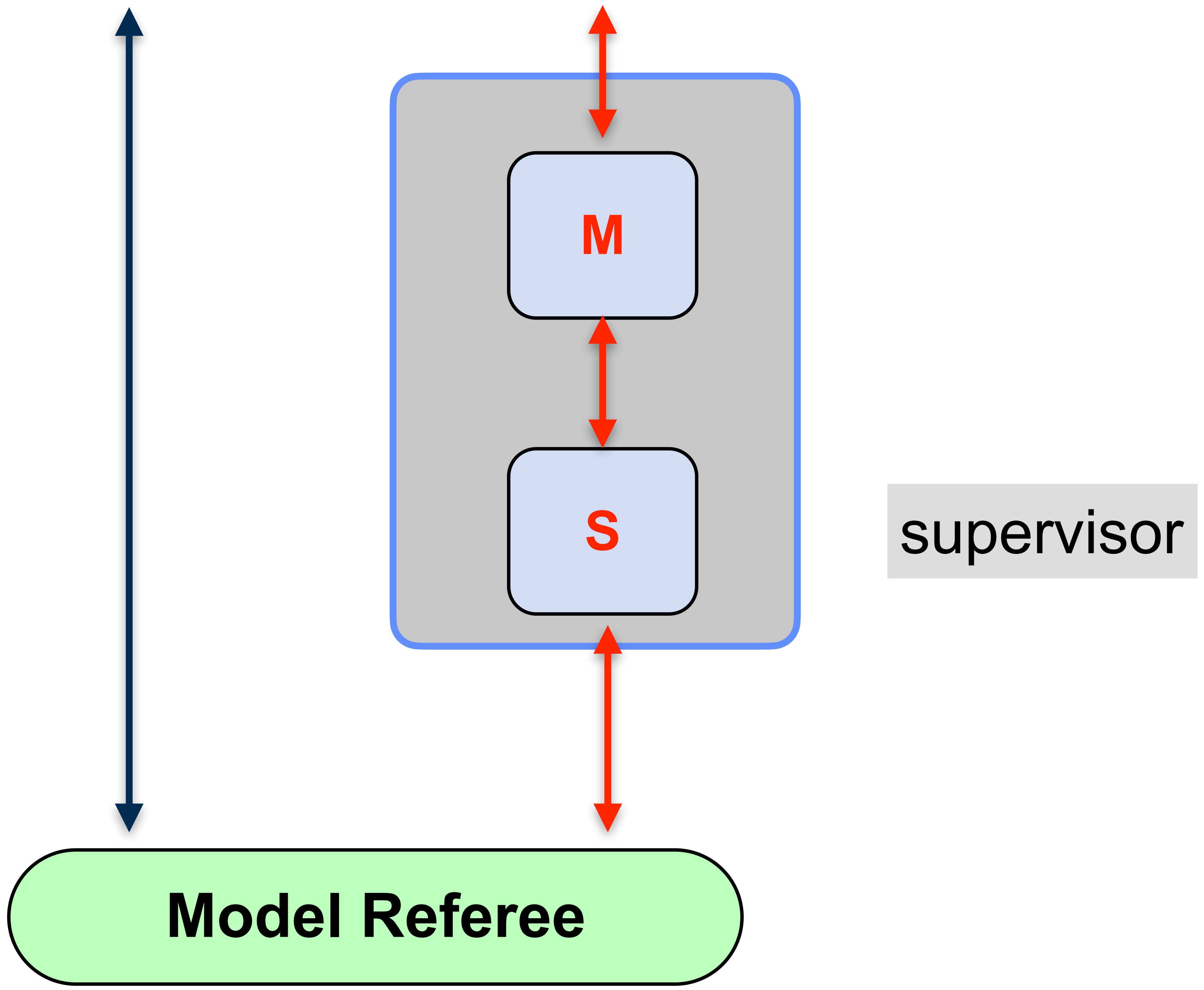
CML + AC Simulator Example



CML + AC: M Commits to a Board

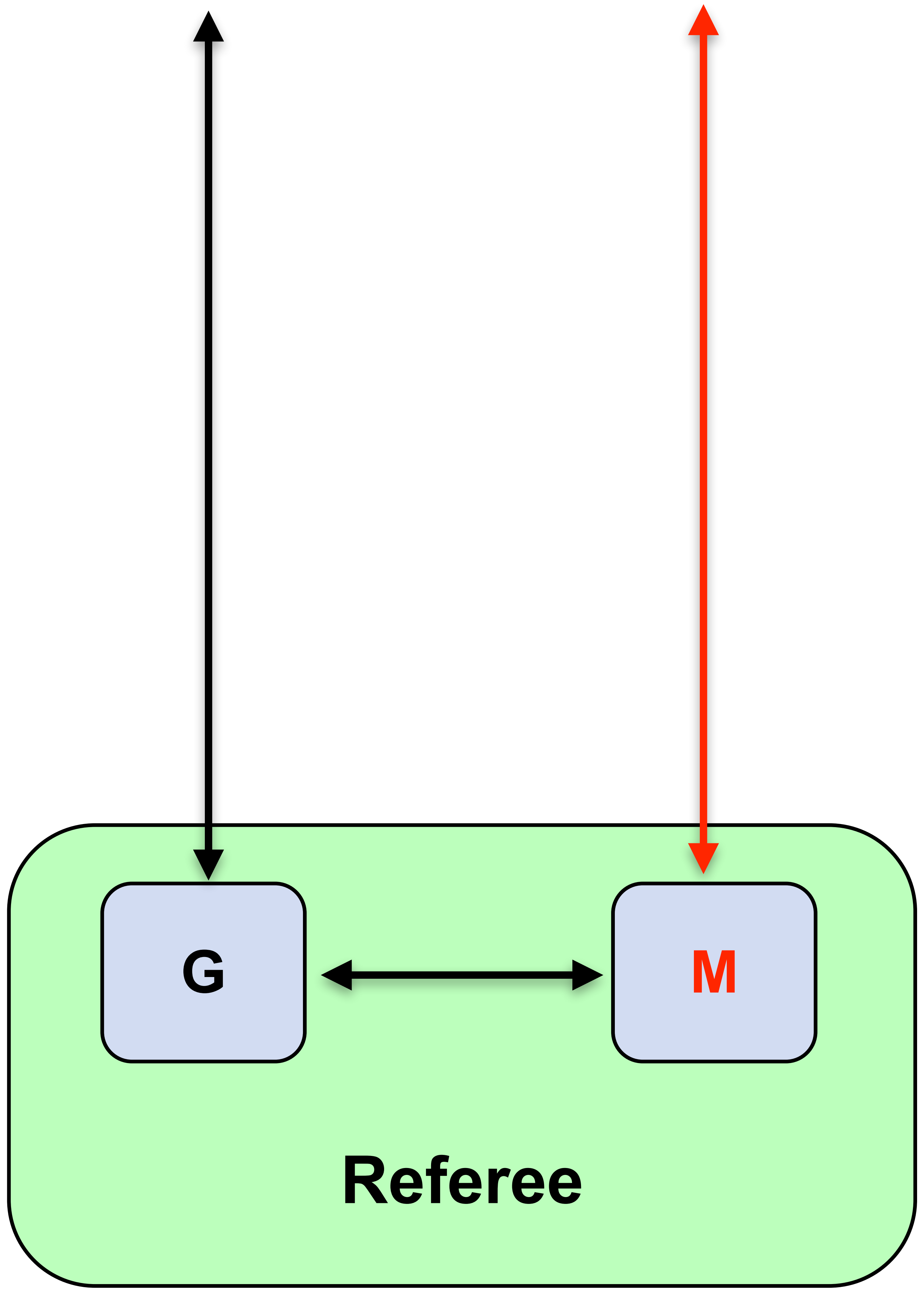


abstract type has *two* kinds of locked boards: one for shooting and one for extraction; S extracts board from locked board M initially provides

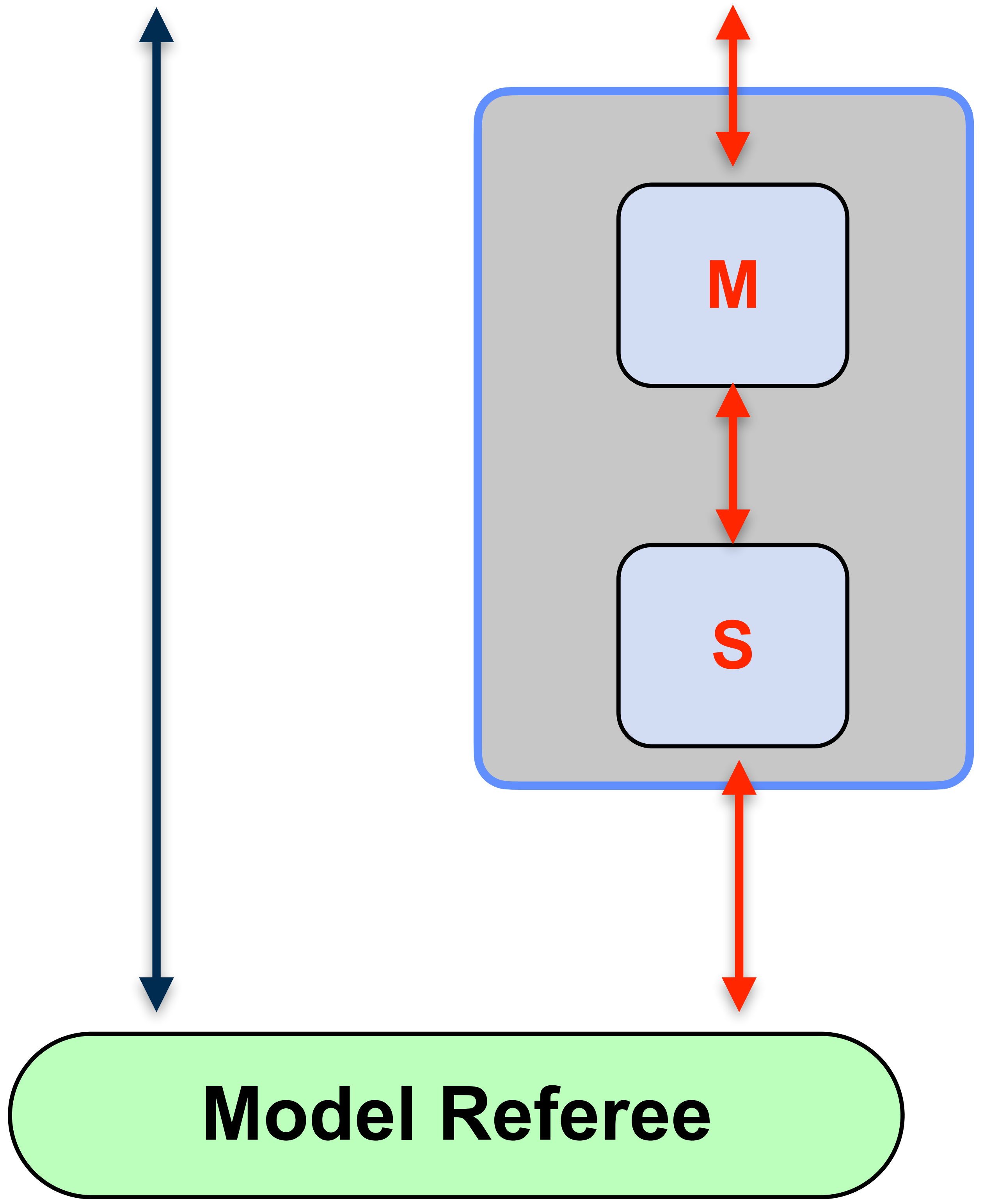


Q: What is the potential pitfall with this approach?

CML + AC: M Commits to a Board



abstract type has *two* kinds of locked boards: one for shooting and one for extraction; S extracts from the locked board M provides its source board, to give to G



supervisor

A: A replay attack in which M gives G back its own locked board must be prevented

Summary

- We used theoretical cryptography's real/ideal paradigm to define when one program interface is secure against a possibly malicious program interface
  - This separates the definition of security from its enforcement
- We gave two secure implementations, using our definition to guide our design and *informally audit* it
  - Using LIO and information flow control
  - Using Concurrent ML + access control
- *We found numerous security bugs during our audits*

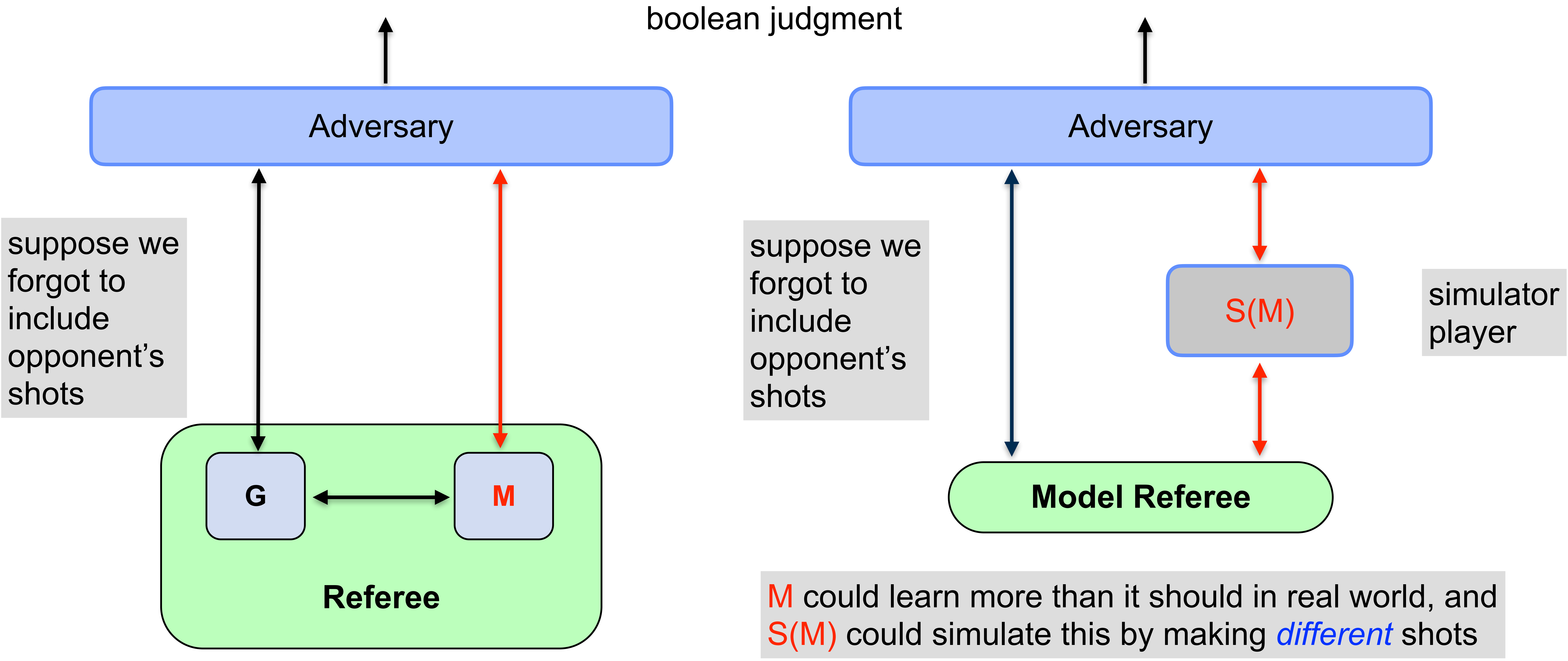
Summary

- Safe Haskell mostly automates the check that the malicious player interface only communicates via its channels
  - But we also want to check that it doesn't do an exit (terminating the whole program) — and this may have to be checked manually
- In Concurrent ML, it must be manually checked that the malicious PI only communicates via its channels

Research Questions

- How do we know that a real/ideal paradigm definition says what we want?
  - Designing ideal functionalities is something of an art, and tools for making their design easier would be useful
  - Tools for helping the designer know they got the correct definition would also be helpful

How Do We Know This Is What We Want?



Research Questions

- What are alternatives to the real/ideal paradigm for defining the security of one component against another?
- When is it useful to split a trusted component into two mutually distrustful ones?
  - For Battleship, are there solutions relying on smaller trusted computing bases?
- When is information flow control necessary to achieve security?
  - Why did Battleship not require information flow control?

Future Work

- We want to prove security using a proof assistant
- It must be possible to formalize and reason about a programming language with
  - A rich module system, supporting abstract types
  - Concurrency
  - Mutable references
- We need to be able to reason about thread scheduling
- We are currently investigating whether the Coq development of the concurrent separation logic Iris would be a good vehicle for this work
  - Joint work with Jared Pincus, Arthur Azevedo de Amorim and Marco Gaboardi

Example 3: Battleship

Questions about Example 3?

Real/Ideal Paradigm Summary and Discussion

- Let’s end these lectures with an open discussion about the real/ideal paradigm
- Possible discussion points:
  - Difficulty defining ideal functionalities capturing correct security notions
  - Approaches to proving security in the real/ideal paradigm
  - Applicability to non-cryptographic security
  - Possible alternative approaches