SystemC/TLM:
AcceLera SystemC 2.3.1

Franco Fummi

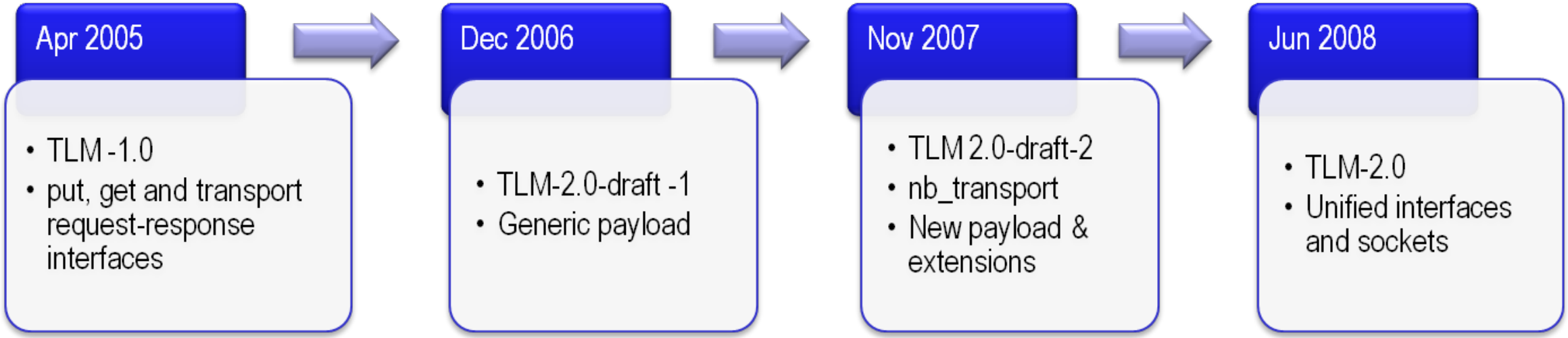
UNIVERSITÀ di VERONA
Dipartimento di INFORMATICA

Version 1.1

Contents

- Introduction
  - History
- Use cases, coding styles and mechanisms
  - Coding styles: Loosely vs. Approximately Timed
  - Initiators and targets
  - Blocking vs. Non-blocking transport
  - The time quantum
- Sockets and generic payload
- Verification principles

OSCI TLM Development



NON SUPPORTATO

History of Releases (I)

- TLM-2005-04-08: TLM 1.0
  - Bidirectional interfaces
    - Transport, blocking
  - Unidirectional interfaces
    - FIFO, blocking, non-blocking
  - Separation between request and response
- TLM-2006-11-29: TLM 2.0 draft 1
  - TLM 1.0 + timing annotation
  - Generic payload + extensions

History of Releases (II)

- TLM-2007-11-29: TLM 2.0 draft 2
  - Legacy support for TLM 1.0
  - Coding styles
    - Loosely-timed, Approximately-timed
  - New Transport interfaces
  - Temporal decoupling
  - Quantum keeper
  - Improved Debug and Direct Memory interfaces
- TLM-2008-06-09: TLM 2.0 standard
- TLM-2009: TLM 2.0.1 (minor changes and bug fixes)
- Standard migrated into ***Accellera SystemC 2.3.1***

Use Cases, Coding Styles and Mechanisms

TLM: Transaction-Level Modeling

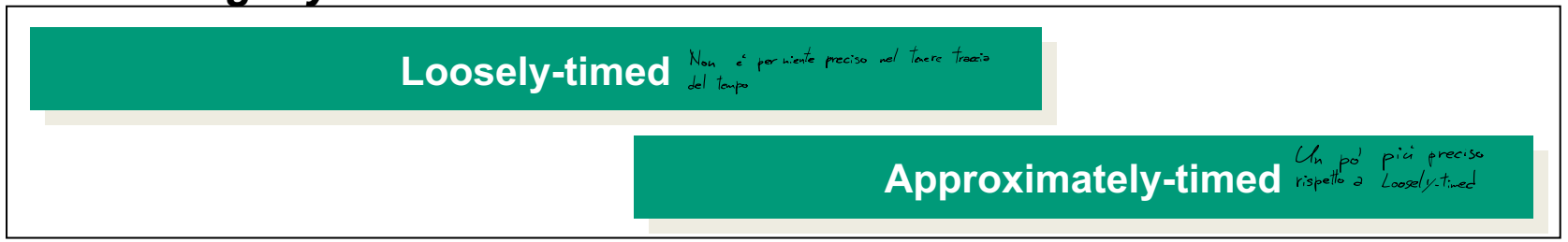
I tempi di simulazione sono molto più bassi, PERÒ non so i cicli di clock esatti per la simulazione ⇒ l'unità di tempo diventa la transazione

Chiedo l'esecuzione di un metodo, mando dati e ricevo altri di output

Use cases

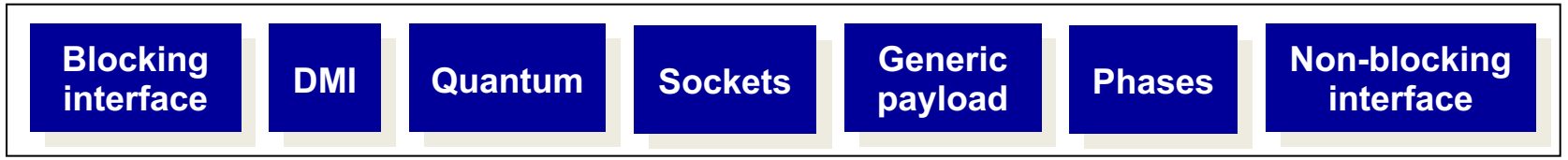
I moduli possono essere esportati/importati

TLM-2 Coding styles



Se non ho bisogno di tempi esatti dell'hw uso TLM che è molto veloce a simulare

Mechanisms → cosa la libreria TLM fornisce

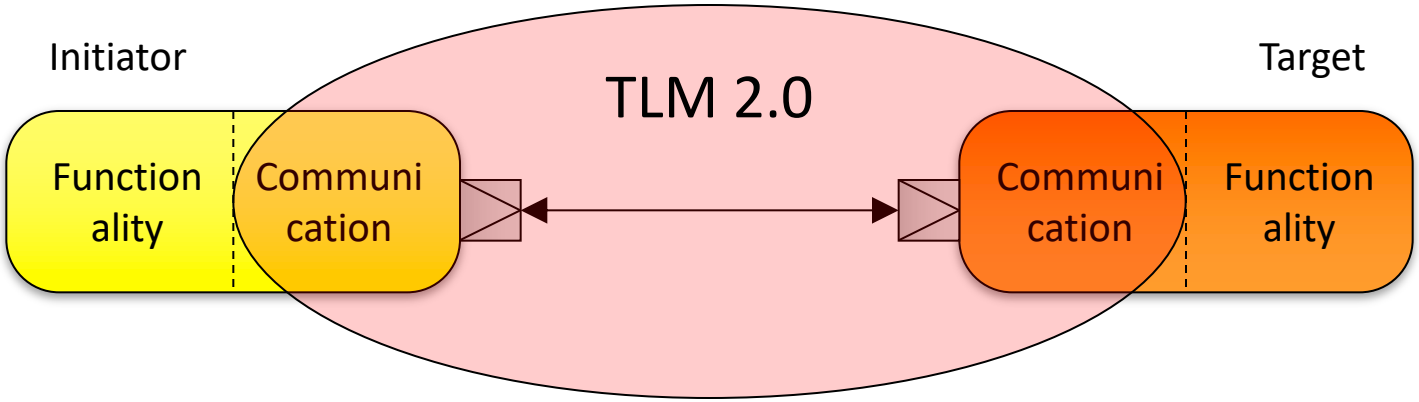


Functionality vs Communication

- TLM manages to keep distinct functionality and communication in each module

Le funzionalità possono essere in qualsiasi linguaggio, basta poterlo collegare a C++

- TLM 2.0 allows to model the communication part (i.e. how each module interacts with the others)



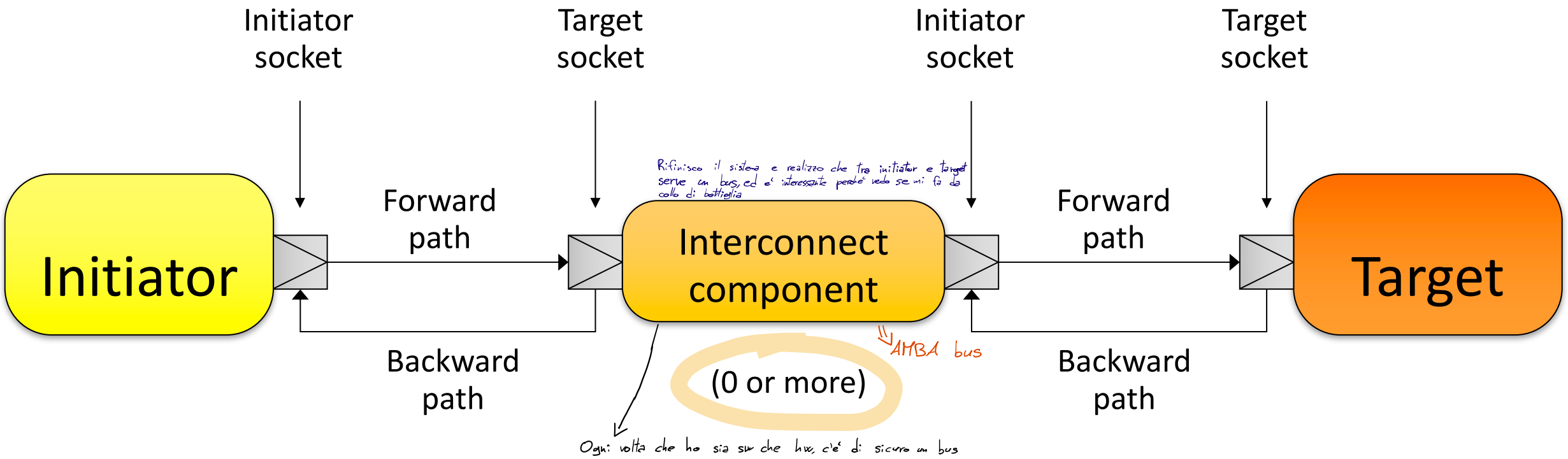
Transactions

- TLM relies on the notion of transaction
- A transaction consists of a data transfer from a design module to another one
  - Write and read operations are examples of transactions
  - It is usually represented by a generic payload object in the code
    - This object contains both data and control information
    - (e.g., address and type of command)
  - It is exchanged between modules through primitive calls

Socket, Initiator and Target (I)

- Communication is achieved by exchanging packets between an initiator module and a target module, through a socket
  - The initiator starts a transaction
  - The target is the end point of a transaction
  - An interconnect component is an intermediate point in the path from the initiator to the target
  - A socket connects two modules, and allows them to communicate by means of the available interfaces
  - The forward path runs from the initiator to the target
  - The backward path runs from the target to the initiator

Socket, Initiator and Target (II)



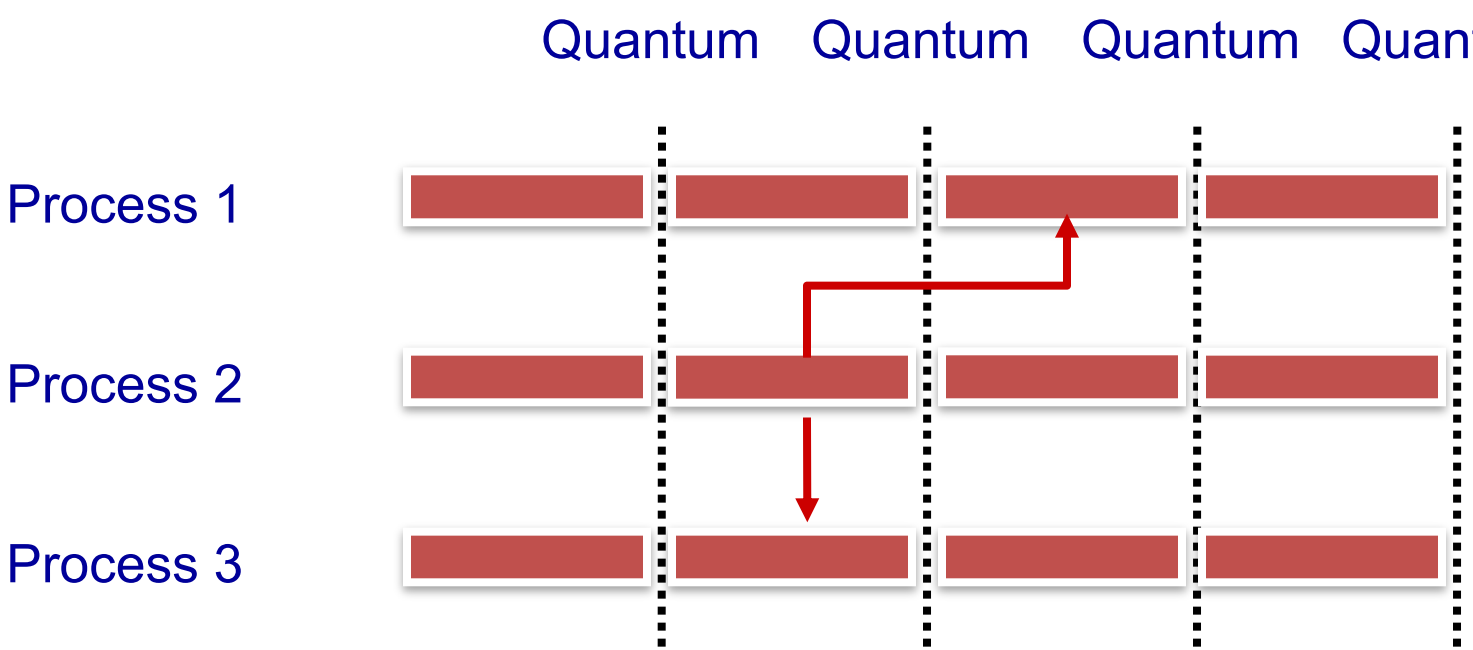
Coding Styles

- A coding style defines how time and data are related together, according to designers' perception
- It is NOT a specific abstraction level
- More of a guideline, designers enjoy several degrees of freedom
- TLM 2.0 standard provides two coding styles
  - Loosely-timed (LT)
  - Approximately-timed (AT)
- It is possible to define other coding styles

Loosely-timed (LT)

- Sufficient timing details to boot an operating system and run multi-core systems
- Allows processes to run ahead of simulation time (temporal decoupling) → faster simulation
- Each transaction has two timing points
  - Start and end of transaction
- Loosely-timed models have a limited number of temporal dependencies and context switches → faster simulation

Loosely-timed



3: proc 1 deve aspettare proc 3
2: proc 3 deve aspettare proc 1

Each process runs ahead up to quantum boundary

sc_time_stamp() advances in multiples of the quantum

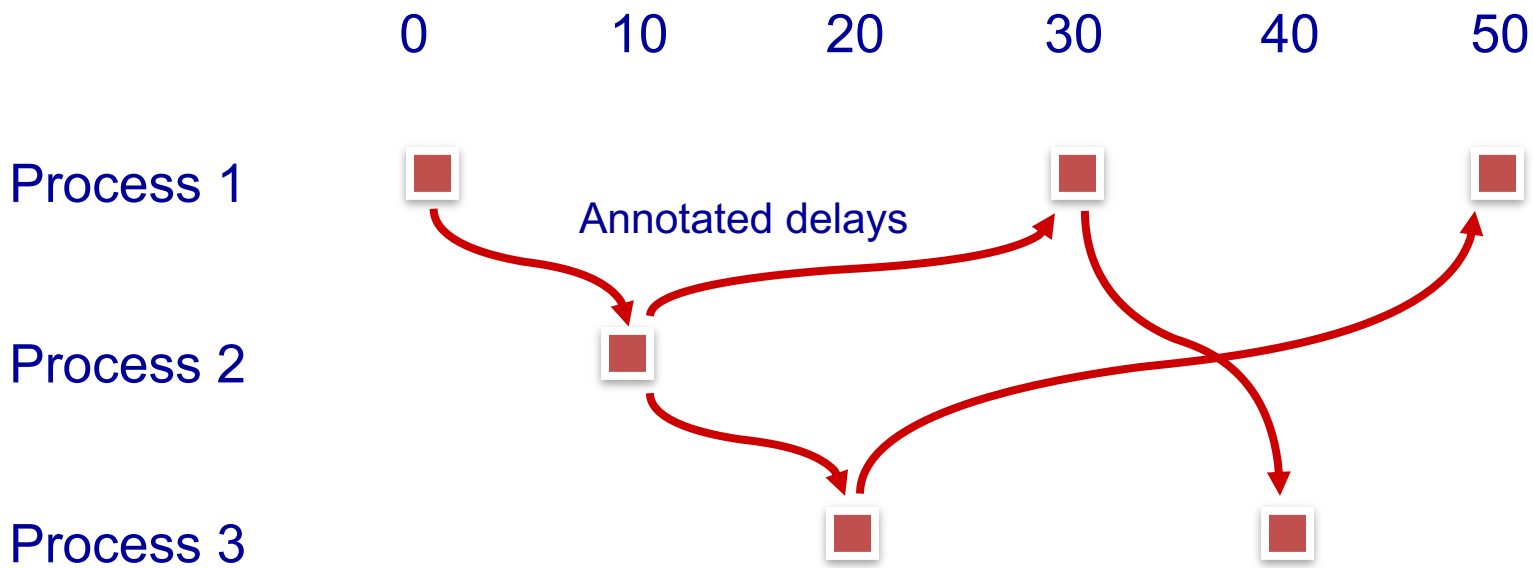
Deterministic communication requires explicit synchronization

Approximately-timed (AT)

- Appropriate for architectural exploration and performance analysis
- Transactions are broken down into multiple phases
  - For example, four phases in the base protocol
    - Beginning of the request (BEGIN_REQ)
    - End of the request (END_REQ)
    - Beginning of the response (BEGIN_RESP)
    - End of the response (END_RESP)
  - It is possible to create specific protocols with a different number of phases
- Processes run in lock-step with simulation time
- Timing accuracy is needed, so process interactions are annotated with specific delays

Se continuo a sincronizzare initiator e target alla fine avrò il tempo esatto

Approximately-timed



Each process is synchronized with SystemC scheduler

Delays can be accurate or approximate

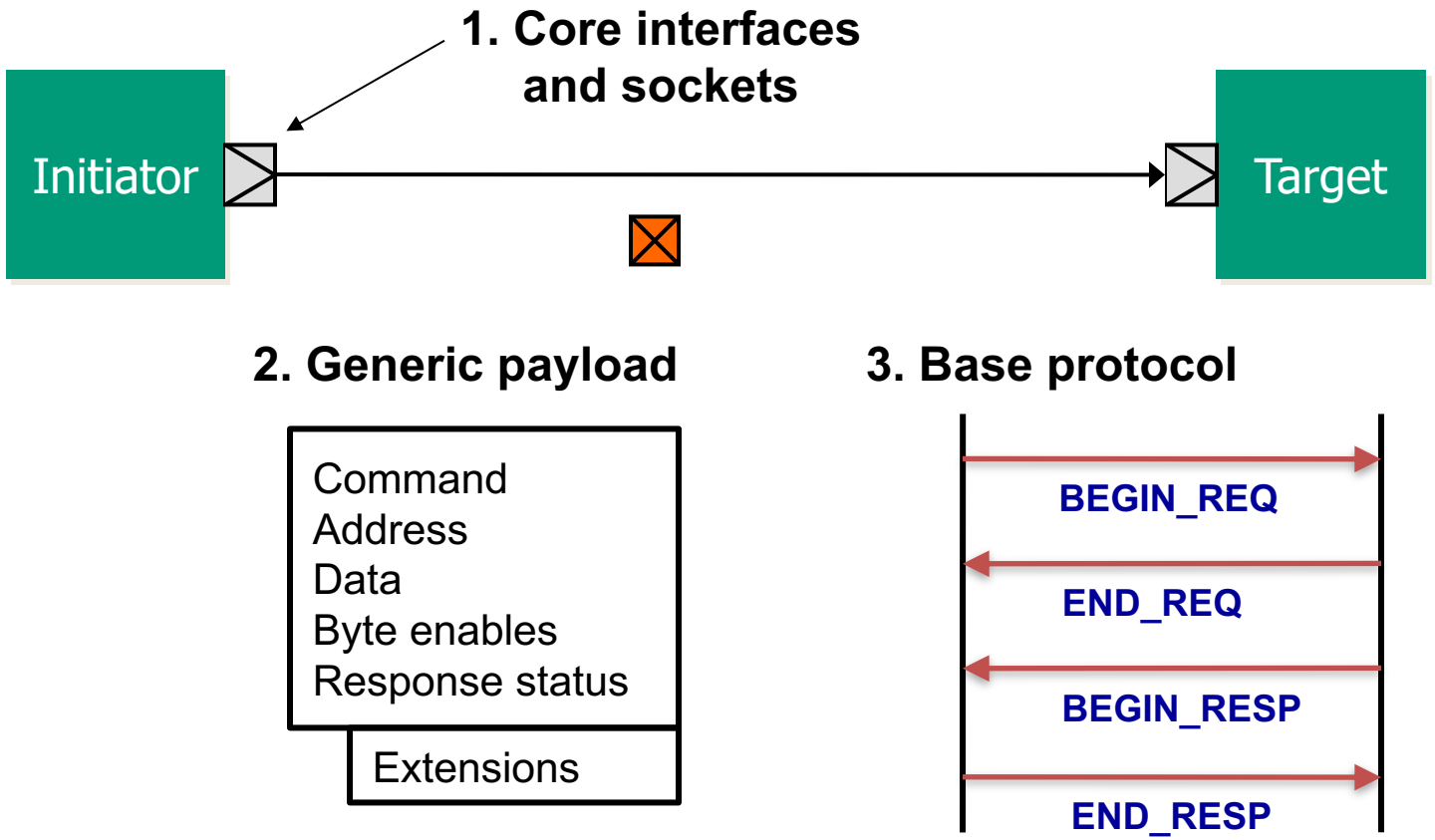
Untimed

- The notion of simulation time is unnecessary, so it is not taken into account
- TLM 2.0 does not explicitly support this coding style, because all contemporary bus-based systems require some notion of time to model software running on embedded processors
- An untimed model may be obtained by removing timing annotation, quantum keeping and temporal decoupling from a loosely-timed model

Interfaces

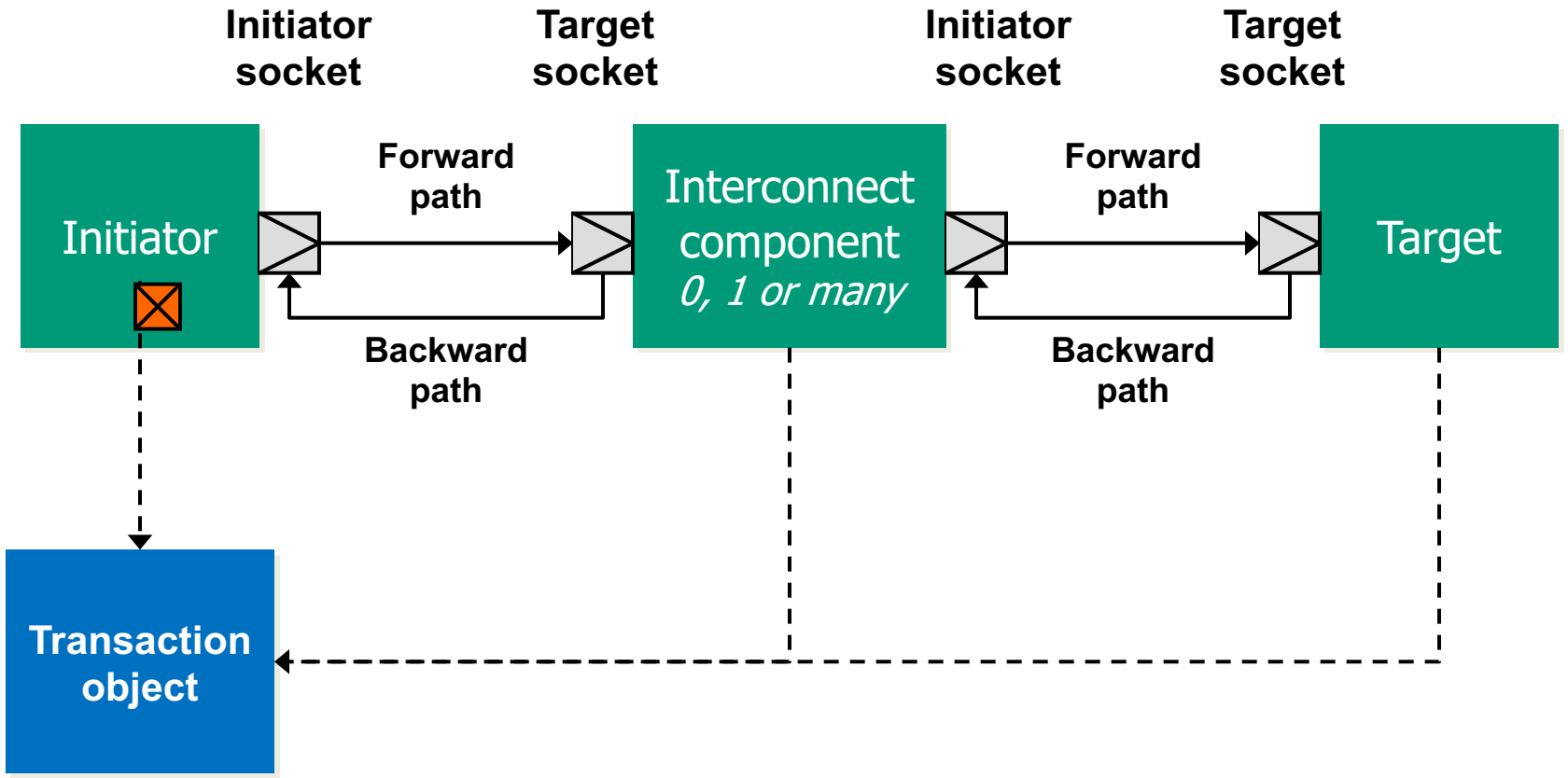
- Blocking transport interface
  - Intended to support the loosely-timed coding style
- Non-blocking transport interfaces
  - Intended to support the approximately-timed coding style
- Direct memory interface (DMI)
  - Allows the initiator to access an area of memory owned by the target by using a direct pointer
- Debug interface
  - Allows the initiator to communicate with the target without delays, waits, event notifications

Interoperability Layer



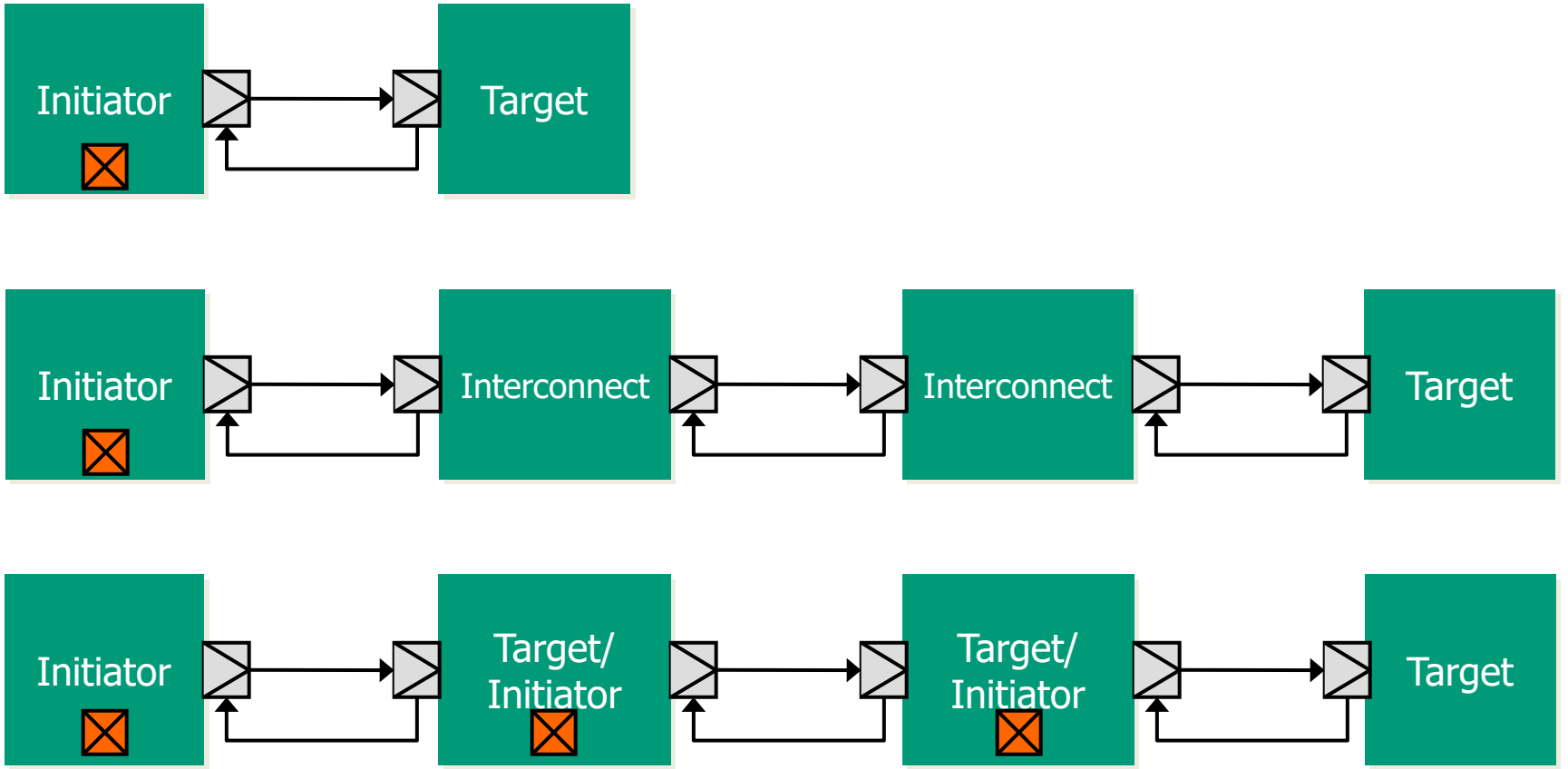
Maximal interoperability for memory-mapped bus models

Initiators and Targets



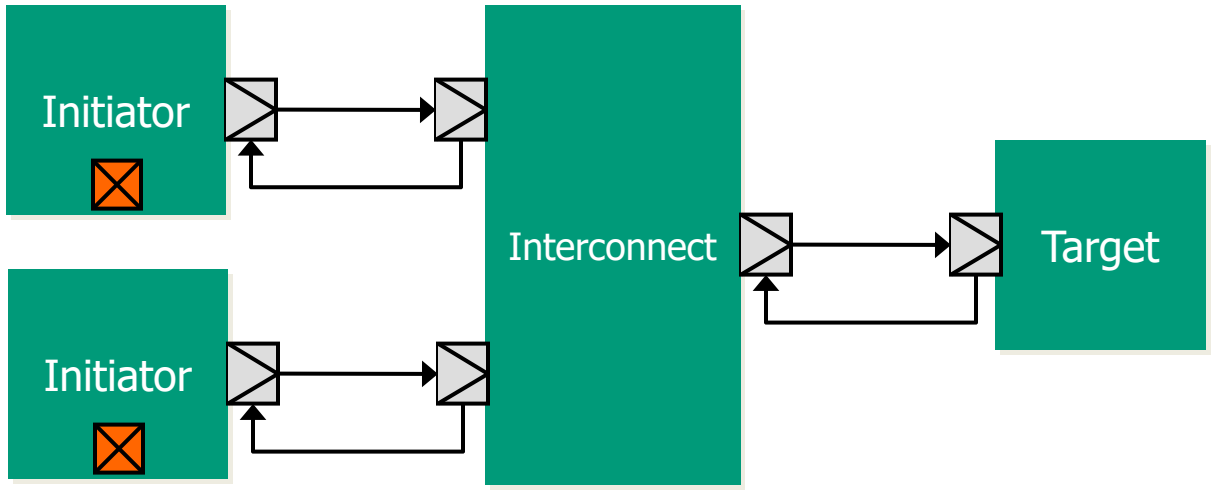
References to a single transaction object are passed along the forward and backward paths

TLM-2 Connectivity



Transaction memory management needed

Convergent Paths

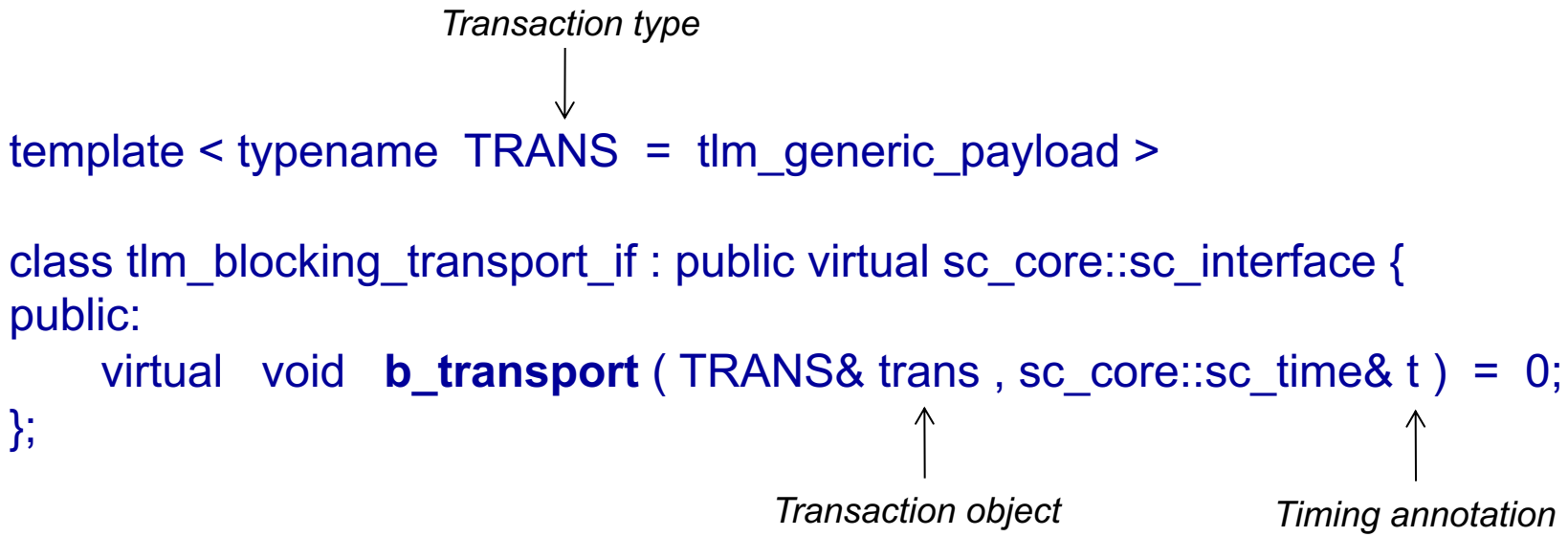


Blocking versus Non-blocking Transport

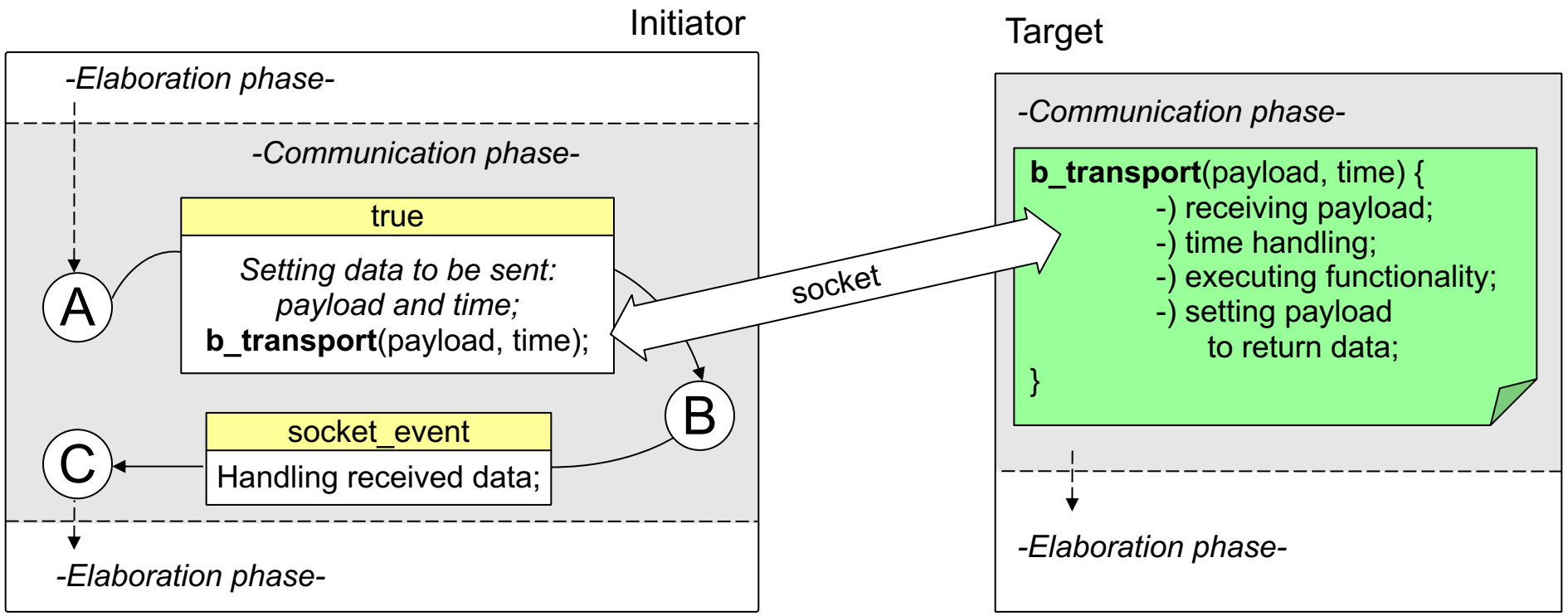
- Blocking transport interface
  - Includes timing annotation
  - Typically used with loosely-timed coding style
  - Forward path only
- Non-blocking transport interface
  - Includes timing annotation and transaction phases
  - Typically used with approximately-timed coding style
  - Called on forward and backward paths
- Share the same transaction type for interoperability
- Unified interface and sockets – can be mixed

Blocking Interface

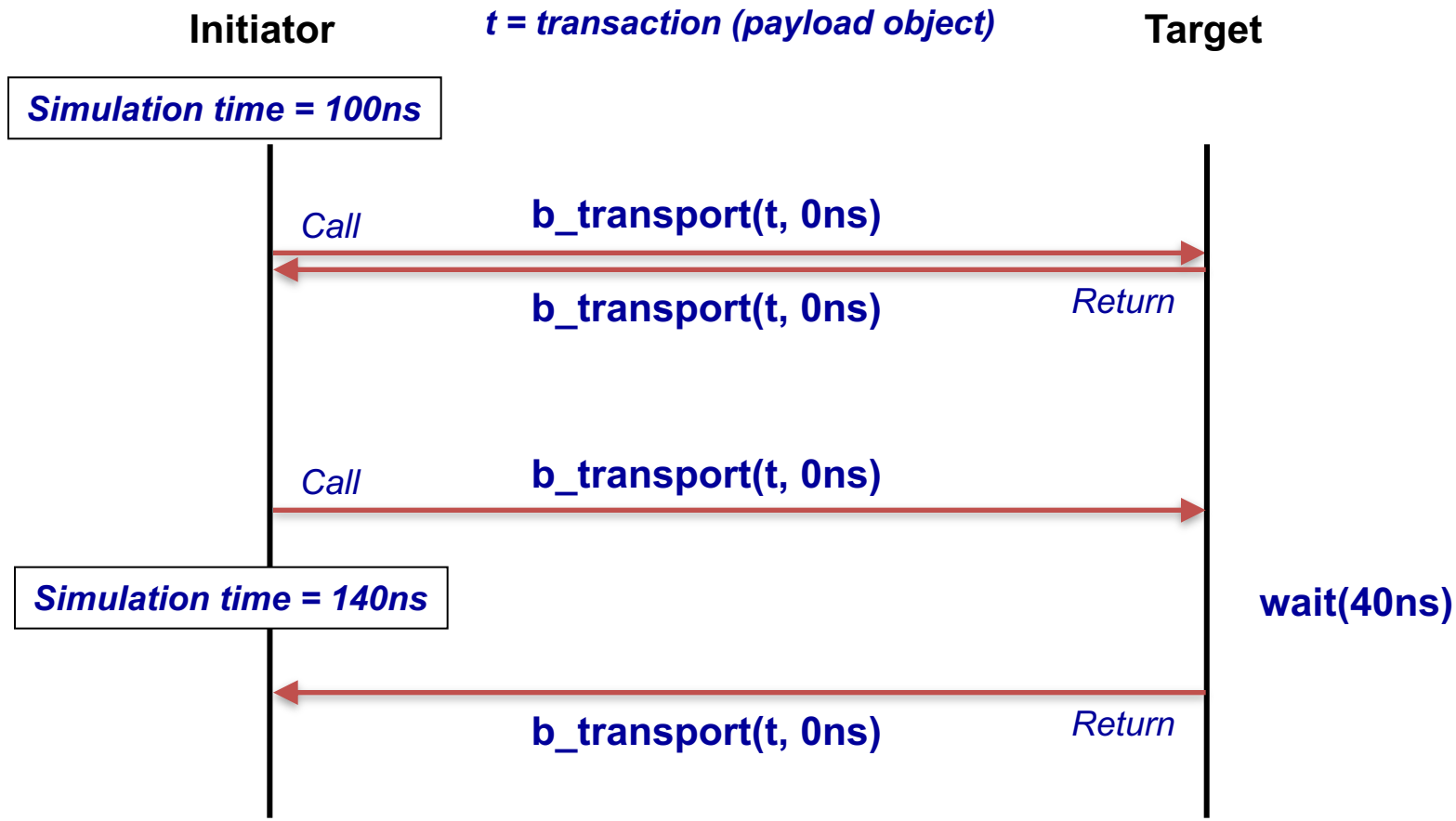
- Appropriate where an initiator wishes to complete a transaction with a target in a single function call
- Two timing points
  - Call to and return from the blocking transport function
- It only uses the forward path from initiator to target



EFSM Model of a LT-based Protocol



Blocking Interface Example



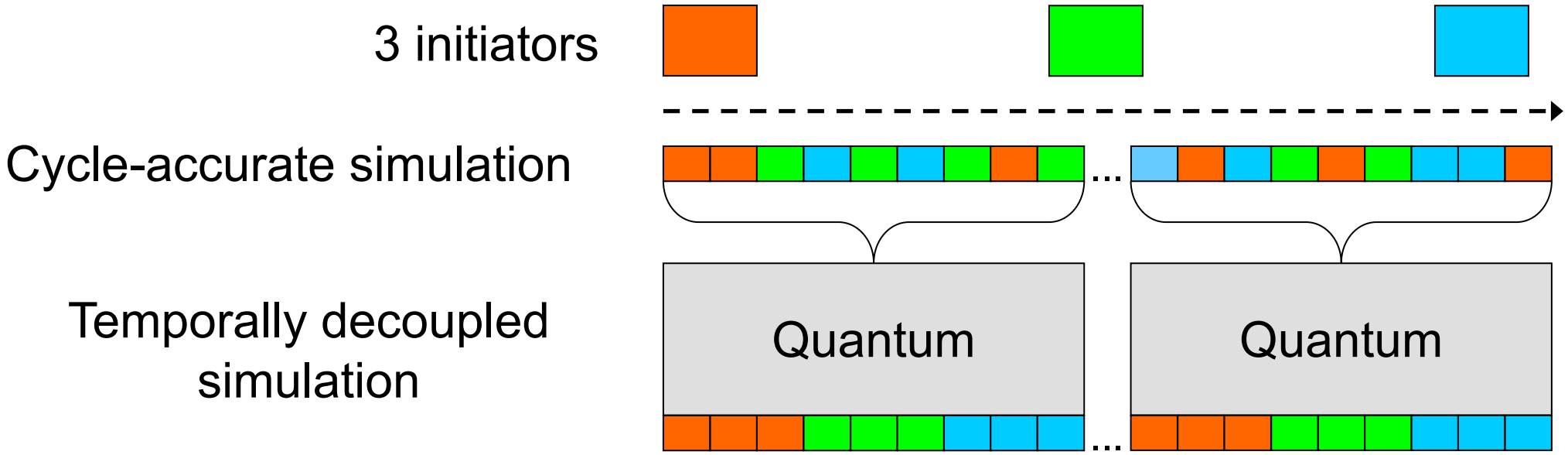
Initiator is blocked until return from b_transport

Temporal Decoupling

- A temporally decoupled initiator may run at a notional local time in advance of the current simulation time
  - in this case it should pass a non-zero value for the time argument to the b_transport() primitive
- Each process is allowed to run for a certain time slice (time quantum) before switching to the next process
- A temporally decoupled initiator will continue to advance local time until the time quantum is exceeded or it reaches an explicit synchronization point
- At that point, the initiator is obliged to synchronize by suspending execution
- Simulation time cannot advance until the initiator thread yields
- The quantum keeper is responsible for enforcing quantum timing

Time Quantum

- Each process runs ahead up to quantum boundary
- Simulation time advances in multiples of the quantum
- Fewer context switches → faster simulation, but also less time accuracy
- Deterministic communication requires explicit synchronization



Quantum Keeper Example (I)

```
struct Initiator: sc_module {
    tlm_utils::tlm_quantumkeeper  m_qk;

    SC_CTOR(Initiator) : init_socket("init_socket")  {
        m_qk.set_global_quantum( sc_time(500, SC_NS) );
        m_qk.reset();
    }
    void thread() { ...
        for (int i = 0; i < RUN_LENGTH; i += 4) {
            local_time = m_qk.get_local_time() ;
            init_socket->b_transport( trans, local_time );
            cout << "After b_transport – time = " << sc_time_stamp()
                 << " + " << local_time << endl;
            m_qk.set( local_time );
            m_qk.inc( sc_time(100, SC_NS) );
            cout << "Initiator completed – time = " << sc_time_stamp()
                 << " + " << m_qk.get_local_time() << endl;
            if ( m_qk.need_sync() ) {
                cout << "### SYNCHRONIZING ###" << endl;
                m_qk.sync();
            }
        } } Local time
    }
};
```

The quantum keeper object

Set the global quantum

Reset the local quantum

Retrieve current local time

Time consumed by transport

Further time consumed by initiator

Check local time against quantum

Synchronize if necessary

i = 0	i = 1
0 + 150 ns	0 + 400 ns
0 + 250 ns	0 + 500 ns
	500 + 0 ns

Simulation time

Non-blocking interfaces (I)

- Appropriate where it is desired to model the detailed sequence of interactions between initiator and target during the course of each transaction
  - Multiple function calls to complete a transaction
- Multiple phases and timing points
  - Each phase transition is associated with a timing point
- It uses both the forward and the backward path
- Two distinct interfaces for use on opposite paths
  - tlm_fw_nonblocking_transport_if
  - tlm_bw_nonblocking_transport_if

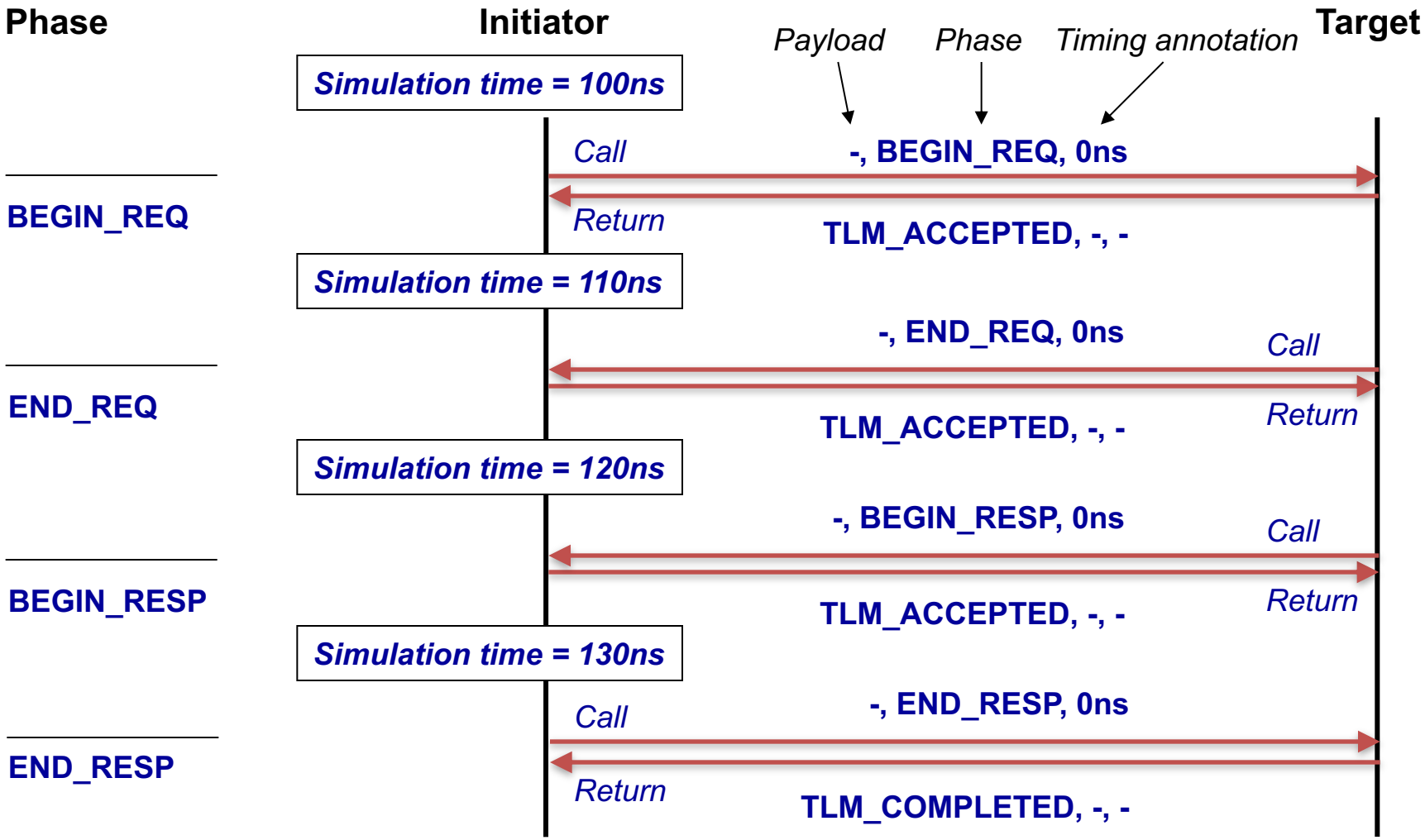
Non-blocking interfaces (II)

```
enum tlm_sync_enum { TLM_ACCEPTED, TLM_UPDATED, TLM_COMPLETED };

template <  typename TRANS = tlm_generic_payload,   <-- Transaction type
            typename PHASE = tlm_phase>              <-- Phase type

class tlm_fw_nonblocking_transport_if : public virtual sc_core::sc_interface {
public:
   virtual tlm_sync_enum nb_transport_fw(   TRANS& trans,
                                            PHASE& phase,
                                            sc_core::sc_time& t ) = 0;
};

class tlm_bw_nonblocking_transport_if : public virtual sc_core::sc_interface {
public:
   virtual tlm_sync_enum nb_transport_bw(   TRANS& trans,   <-- Transaction object
                                            PHASE& phase,   <-- Phase object
                                            sc_core::sc_time& t ) = 0;
};
                                                  ^
                                            Timing annotation
```

tlm_sync_enum overview

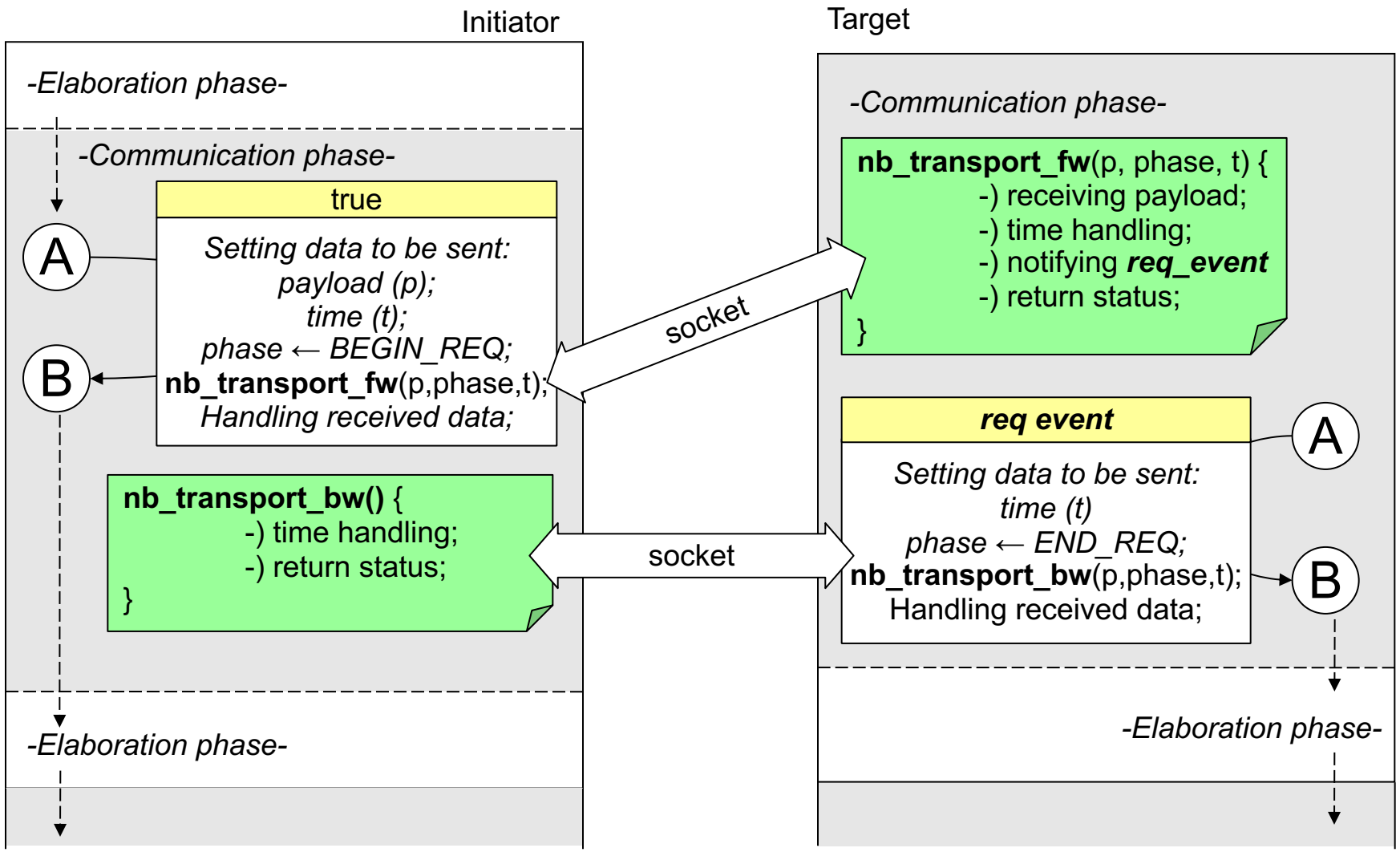
- TLM_ACCEPTED
  - The callee shall not have modified the transaction object, the phase or the time argument during the call
- TLM_UPDATED
  - The callee has updated the transaction object
  - It may have modified the phase argument
  - It may have increased the value of the time argument
- TLM_COMPLETED
  - The callee has updated the transaction object
  - It may have increased the value of the time argument
  - The transaction is complete

Non-blocking Interface Example (I)

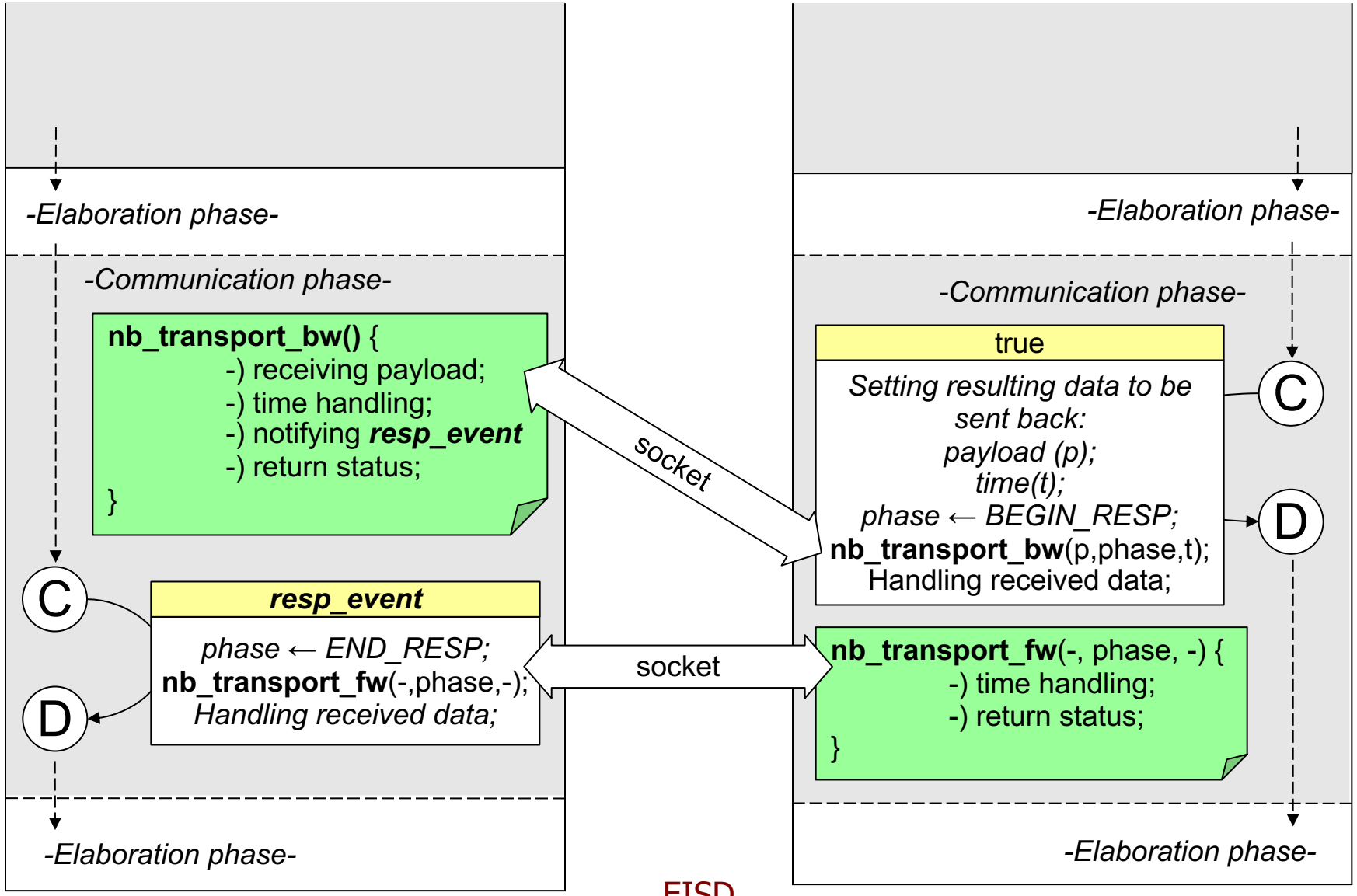


Transaction accepted now, caller asked to wait

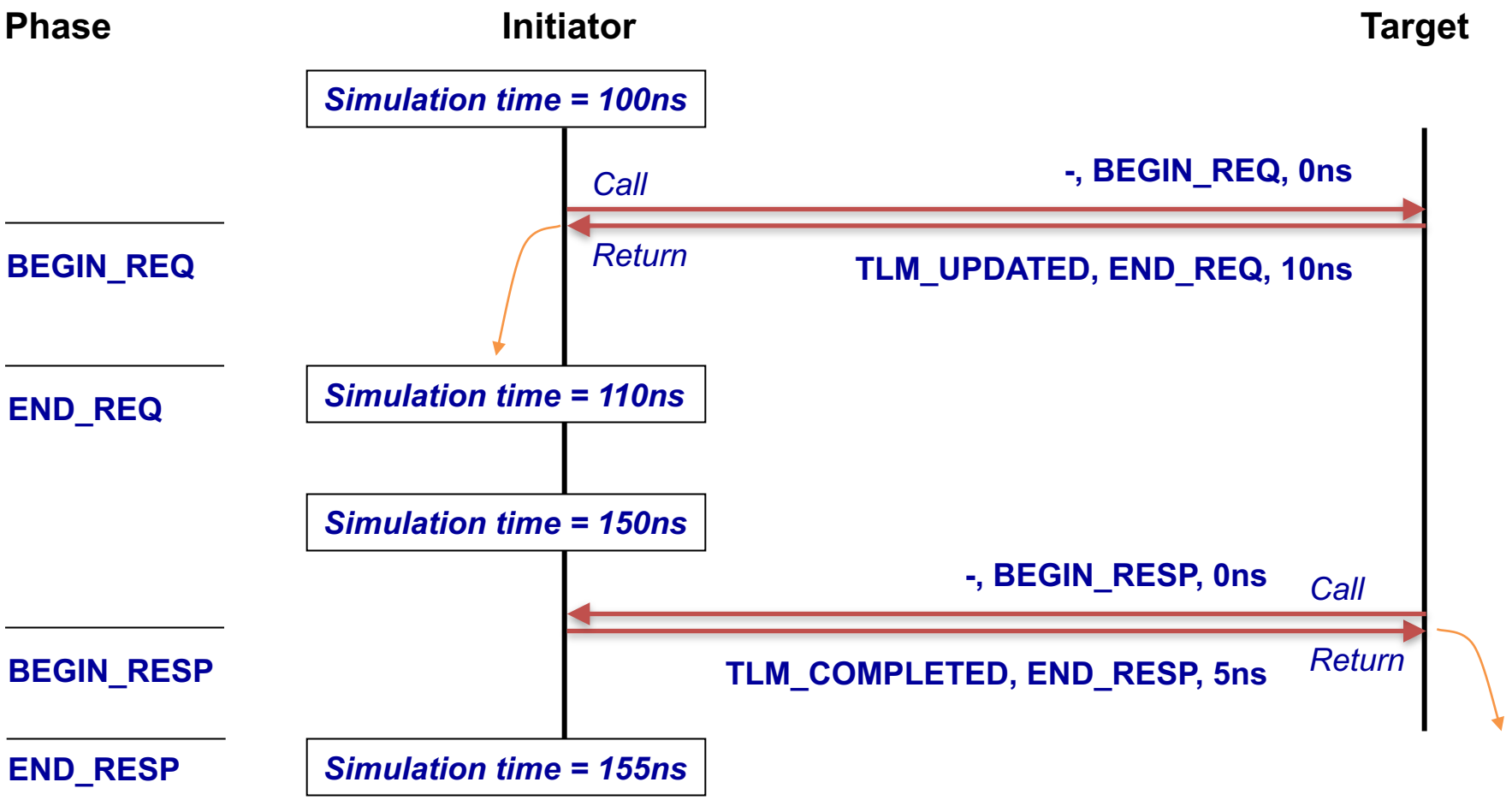
EFSM Model of an AT-based Protocol (I)



EFSM Model of an AT-based Protocol (II)



Non-blocking interface example (II)



Callee annotates delay to next phase transition, caller waits

The Generic Payload

Attribute	Type	Modifiable?	
Command	**tlm_command**	**No**	
Address	**uint64**	**Interconnect only**	
Data pointer	**unsigned char***	**No (array – yes)**	*Array owned by initiator*
Data length	**unsigned int**	**No**	
Byte enable pointer	**unsigned char***	**No (array – yes)**	*Array owned by initiator*
Byte enable length	**unsigned int**	**No**	
Streaming width	**unsigned int**	**No**	
DMI hint	**bool**	**Yes**	
Response status	**tlm_response_status**	**Target only**	
Extensions	**(tlm_extension_base*)[ ]**	**Yes**	*Consider memory management*

↳ Puntatori ad altre strutture dati

Response Status

enum tlm_response_status	Meaning
TLM_OK_RESPONSE	**Successful**
TLM_INCOMPLETE_RESPONSE	**Transaction not delivered to target. (Default)**
TLM_ADDRESS_ERROR_RESPONSE	**Unable to act on address**
TLM_COMMAND_ERROR_RESPONSE	**Unable to execute command**
TLM_BURST_ERROR_RESPONSE	**Unable to act on data length or streaming width**
TLM_BYTE_ENABLE_ERROR_RESPONSE	**Unable to act on byte enable**
TLM_GENERIC_ERROR_RESPONSE	**Any other error**

Key Payload Fields

- Command
  - indicates the type of operation (read, write or none) requested to the target
- Address
  - indicates the address in target's memory of data to be read or written
- Data pointer
  - pointer to the data array
    - in case of a read operation, the target will copy the content of a local array in the data array
    - in case of a write operation, the target will copy the content of the data array in a local array
- Data length
  - provides the number of bytes to be stored in or read from the data array
- Response status
  - indicates the outcome of the operation performed by the target

Generic payload extensions

- The generic payload provides an off-the-shelf general purpose payload for abstract bus modeling
  - It is aimed at modeling memory-mapped buses
- To model other specific protocols, the generic payload includes an extension mechanism, so that specialized attributes can be added
  - Generic payload has an array-of-pointers to extensions
    - One pointer per extension type
    - Every transaction can potentially carry every extension type
  - This mechanism allows greater flexibility

System Verification

Step iniziale di verifica tra descrizione e prima versione del sistema

- Validation:
  - Does the design correctly work?
  - Are there implementation design errors?
  - No reference models!
- Verification:
  - Refinement (or abstraction) step verification: ascertaining that system functionality is kept unaltered throughout its representation hierarchy
  - Golden Model as reference

=> SISTEMA IDEALE PER COMPARARE IL MIO SISTEMA ED ESSERE SICURO CHE FUNZIONI

System Verification Techniques

Dynamic verification (simulation)

- ↑ High performance
- ↑ Advanced commercial tools
- ↓ No-exhaustive technique

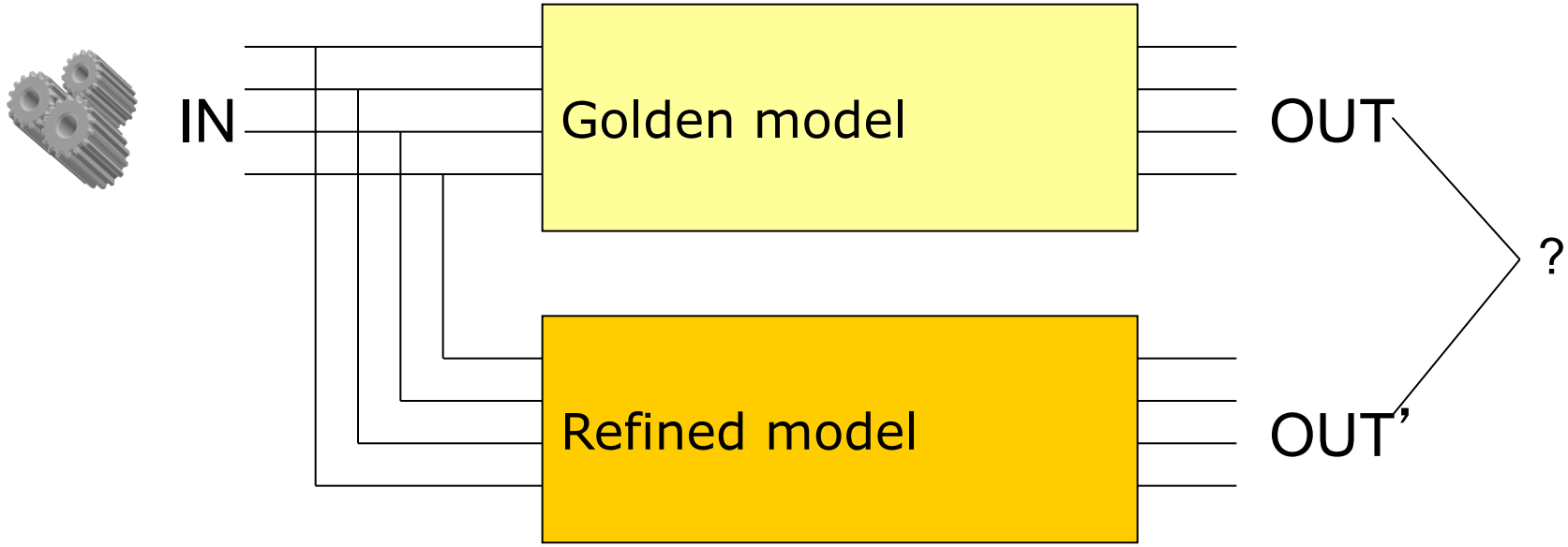
Static verification (formal)

- ↑ Quality
- ↑ Exhaustive technique
- ↓ Long verification time
- ↓ For small size designs

Semi-formal verification

STATO DELL'ARTE ATTUALE

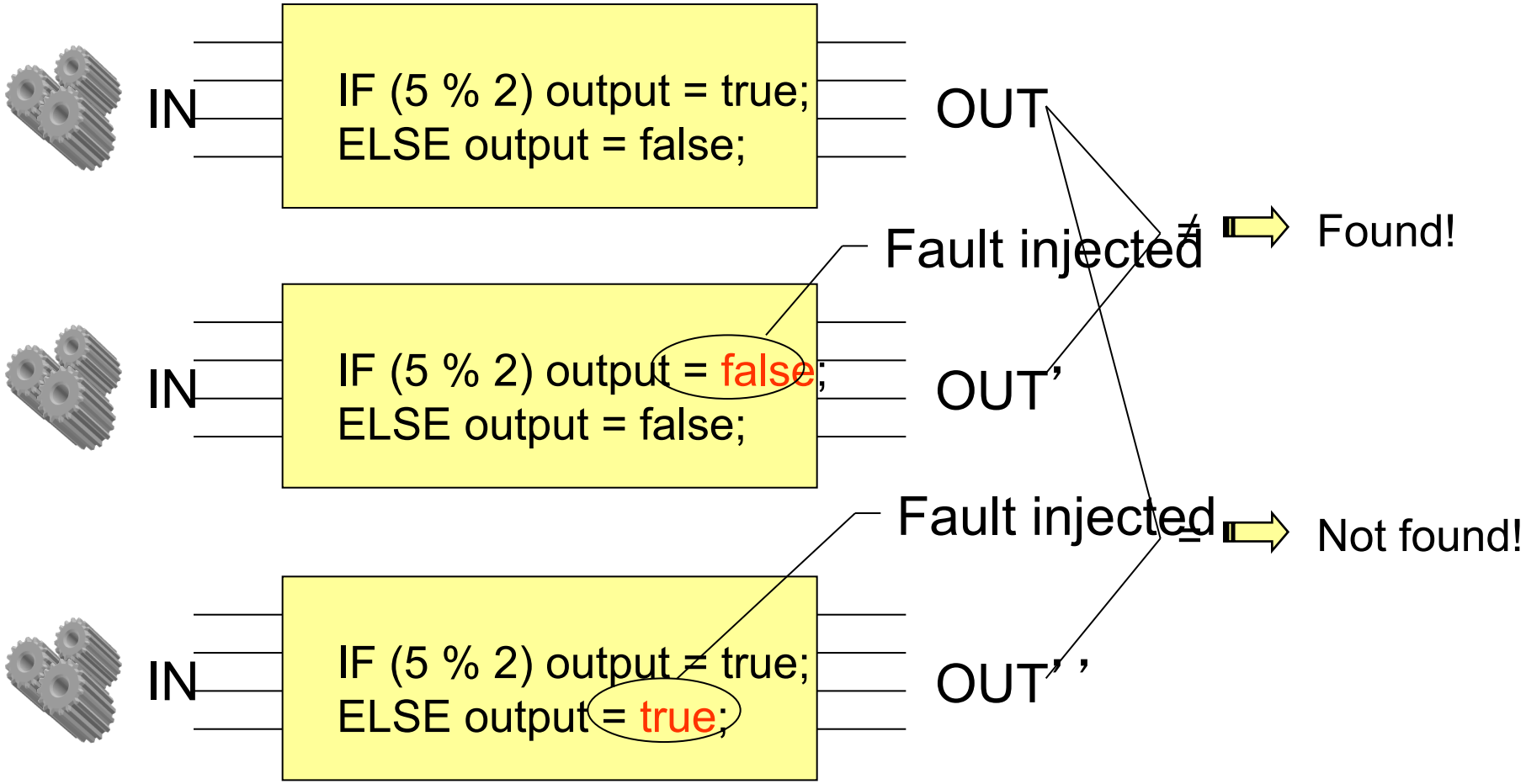
Dynamic Verification (simulation)



- System level and block level verification
- Easiest
- Suitable for TLM & RTL

- Results quality

Dynamic Verification: Faults Injection



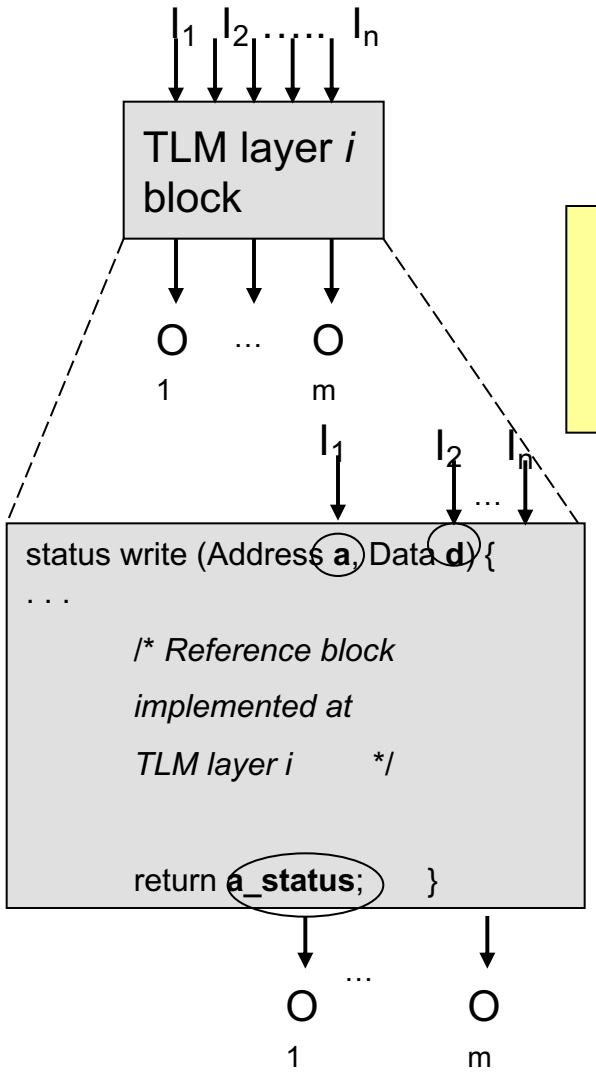
Block level validation

Results quality

Suitable for TLM level 1 & RTL

Synthesizable design required for faults injection

Property Specification Language (PSL)

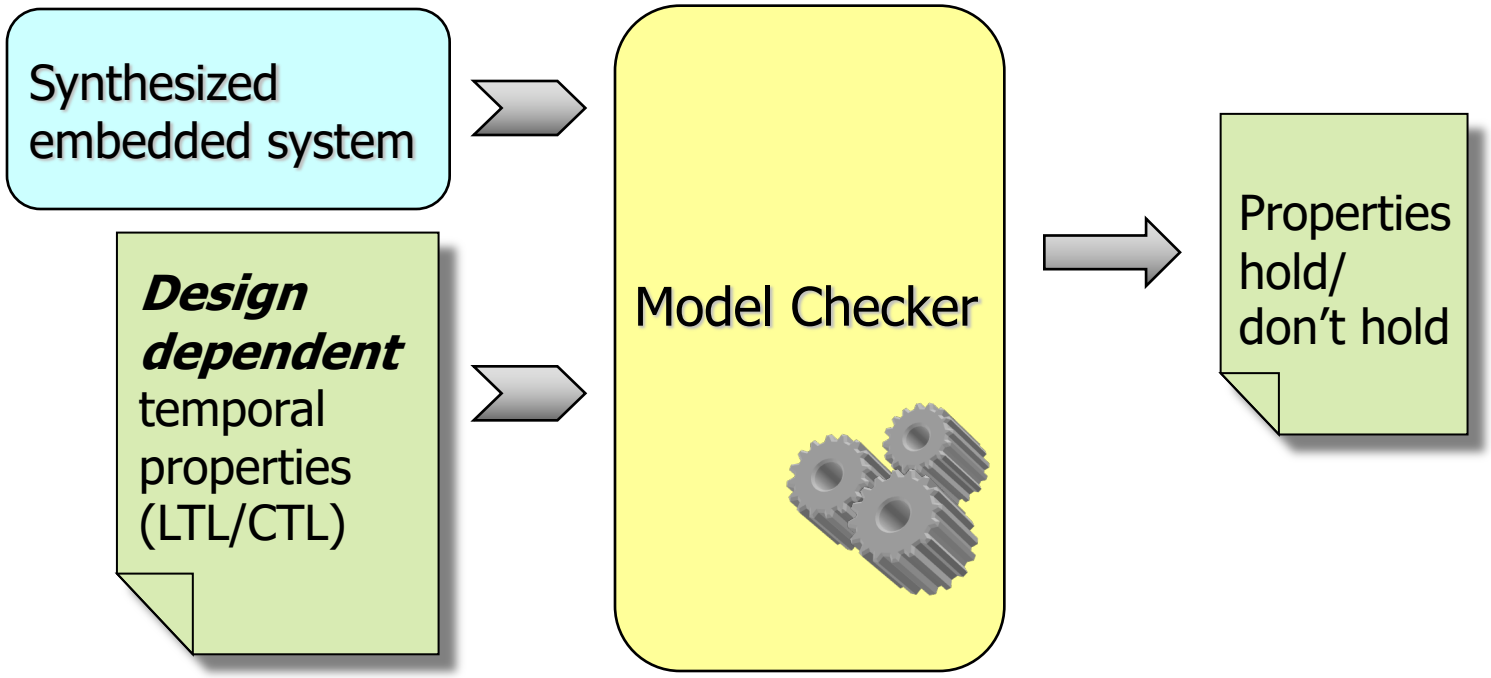



```
// PSL property:
//@PSL always ((a ≥ LOW  &&  a ≤ HIGH  &&  d == DATA )
          => eventually (a_status == SUCCESS));
```

- ⇧ Validation & Verification
- ⇧ Good for dynamic approach
- ⇧ System level and block level
- ⇔ Quite good for formal approach
- ⇧ Suitable for TLM & RTL

Transactor: modulo che traduce i dati da un transaction level ad un altro
Es: risultato prodotto in no time e altro prodotto in funzione di tot cicli di clock

Model Checking



⟺ Block level validation

⇑ Results quality

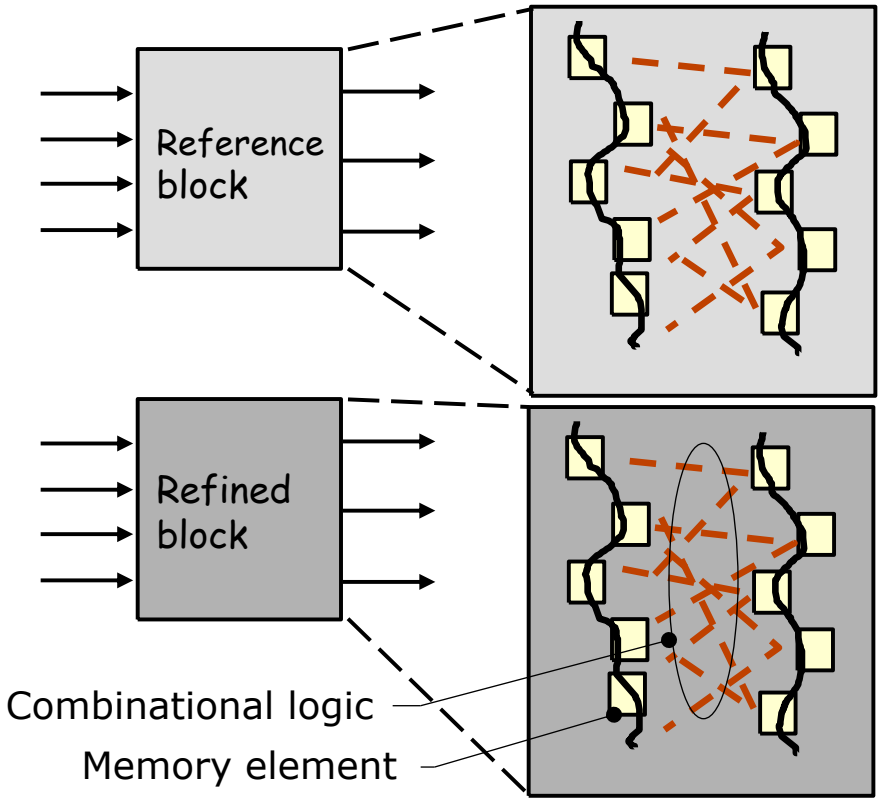
⟺ Suitable for TLM level 1 & RTL

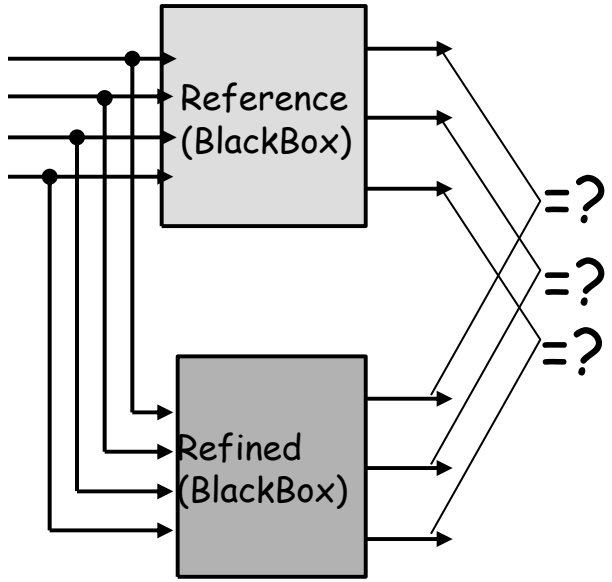
⇓ Synthesizable design required

Equivalence Checking (EC)

Combinational EC

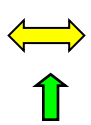
Sequential EC





Equivalence Checking (EC)

- Combinational Equivalence Checking



Block level verification

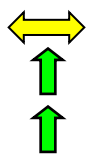
Results quality

Synthesizable design required

Suitable only for RTL vs Gate level verification

For small size designs

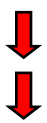
- Sequential Equivalence Checking



Block level verification

Suitable for TLM & RTL

Results quality



Synthesizable design required

For very small size design

Conclusions

- OSCI TLM 2.0 Standard (Accellera SystemC 2.3.1) provides:
  - Transport interfaces with timing annotations and phases
  - Loosely-timed coding style + temporal decoupling for simulation speed
  - Approximately-timed coding style for timing accuracy
  - Generic payload for memory-mapped bus modeling
  - Base protocol to enable interoperability between models
    - Interoperability allows reuse → reduction of production time and costs
  - Extensions for flexibility of modeling