Updated TLM Proposal

Adam Rose
Stuart Swan
John Pierce
29 March 2004

High Level Goals

- First, some high level goals and requirements :
  - Develop a common way of modeling concurrent systems at various abstraction levels so that a "recipe" can be built around these common mechanisms that allows new users to get started and become productive quickly.
  - Promote software safety and help users avoid problems with concurrency, memory leaks, pointer aliasing, SEGFAULTS, etc. (Note that such problems become more severe as more concurrency is represented in the system).
  - Achieve efficiency in space and time without sacrificing code clarity and safety.
  - Provide a set of interfaces that map cleanly and efficiently to both HW and SW, and also cleanly and efficiently across HW/SW boundaries
  - Create interfaces and methods that support IP reuse:
    - within a project from one abstraction level to another
    - across projects
  - This is leads to a design style that encapsulates functions in a polymorphic way so that operations on classes are expressed in ways that are independent of the class. e.g. an arbiter that is independent of the transaction data type.
  - Promote IP interoperability through standard interfaces.

General Strategy

cadence

- Establish a fundamental set of generally useful SystemC TLM interfaces classes.
  - Note that *interfaces* (not channels, ports, processes, etc) are the key constructs in SystemC for constructing modular and reusable blocks. Interfaces are like contracts between different parts of the system.
  - Establish sufficiently rigorous definitions for the TLM interfaces so that code can use the methods defined in the interface without making any additional assumptions about the underlying implementation of those methods.
- Show how these interfaces can be used to solve real problems, e.g. routers and arbiters.
- Show how these interfaces can be used in PV, PVT or CC models and how they can be used to refine from one level of abstraction to the next.

TLM Interface Style

cadence

- The TLM Interface style is the same as sc_fifo, SystemC as a whole, and other C++ libraries eg stl
  - Inbound Data is always passed by const &
    - eg bool nb_put(const &)
  - Outbound Data returned by value if we can guarantee that there will be data to return
    - eg T get(tlm_tag<T> *)
  - If we cannot guarantee that data will come back, we return the status and pass in a non const & :
    - eg bool nb_get(T &)
  - We never use pointers
  - We never use non const & for inbound data
- While this is not pure pass-by-value, it shares the need to provide copy constructors and destructors with pass-by-value

Usability, Safety and Speed

- There is always a three way trade off between speed, safety and usability.
- Usability
  - Because this style is very widespread and intuitive, it is easy to learn
- Safety is guaranteed if
  - The master owns the data *or*
  - We are using a memory management system such as boost::shared_ptr *or*
  - We take a copy of any inbound data before the first wait
- Speed
  - We use const & for inbound data
  - We can create a vector<T> class for example that has copy-on-write semantics and which is very efficient to pass by value.

Terminology : Blocking vs Non Blocking

cadence

TLM WG

- Blocking
  - "Blocking" means transaction is completed in *one* function call eg

  transport(transaction &)
- Non blocking means
  - We need *more than one* function to complete the transaction eg :

  port->send_request(req)

  rsp = port->get_response();
- Wait
  - We make *no assumptions* about whether the function does or does not have waits in the implementation
- Implementation
  - There is an implicit assumption that blocking means "implemented in the slave via *sc_export*"
  - There is an implicit assumption that non blocking means "implemented in a *channel*"

OSCI

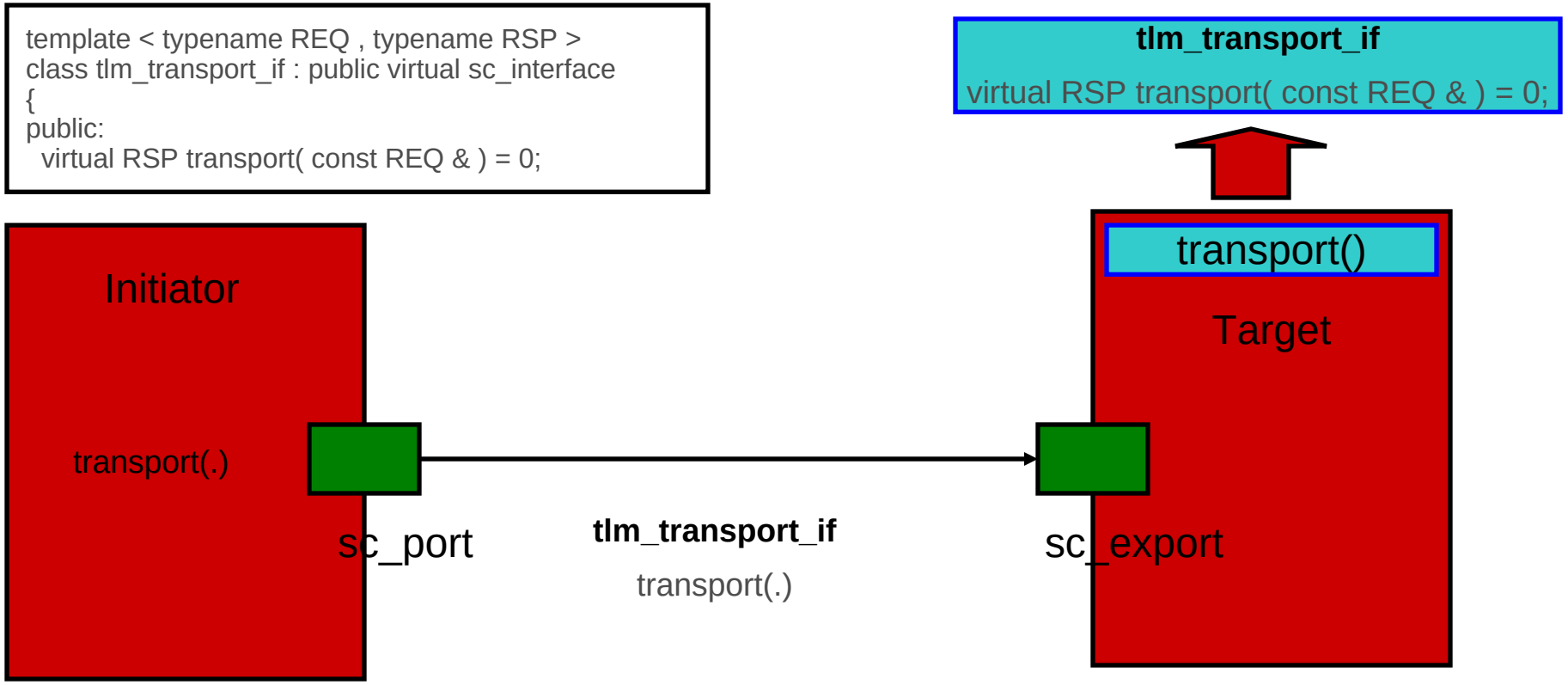
- Blocking
  - Blocking means that this function *must* be called from an SC_THREAD
  - Specifically, there *may be waits* in its implementation
- Non Blocking
  - Non Blocking means that this function can be called from an SC_METHOD or an SC_THREAD
  - Specifically, there must be *no wait's* in the implementation of this function
- Transaction
  - We make *no assumptions* about how many function calls are required to complete a transaction.
- Implementation
  - The terminology applies to *interfaces only*
  - *No assumption* is made re: whether the interface is implemented in a channel or via sc_export

There is no way to avoid the requirement to distinguish between "OSCI blocking" and "OSCI non-blocking" for all methods in all interface classes

When it comes to OSCI standards, OSCI terminology should be used

TLM transport – Basic Architecture

```
template < typename REQ , typename RSP >
class tlm_transport_if : public virtual sc_interface
{
public:
  virtual RSP transport( const REQ & ) = 0;
```



- sc_export **exports** tlm_transport_if for use by outside world
- RSP transport(const REQ &) is **implemented** in the slave
- tlm_transport_if is the tlm bidirectional blocking interface

TLM transport Architecture

master

prot_initiator_port p;

p. read(a , d);

USER LAYER

prot

Slave

my_status read(Address a , Data &d);

my_status write(Address a , Data d);

prot_initiator_port

```
REQ req;
RSP rsp;
my_status read( Address a , Data &d )
{
  req.a = a;
  req.type = read;
  rsp = transport( req );
  d = rsp.d;
  return rsp.status;
```

prot

PROTOCOL LAYER

tlm_transaction

prot_transaction_interface

```
virtual my_status read( Address a , Data &d ) = 0;
virtual my_status write( Address a , Data d ) = 0;
RSP transport( const REQ &req )
{
    if( req.type == read )
        return read( req.addresss , rsp.data );
    else if( req.type == write )
        return write( req.address , req.data );
```

sc_port<REQ,RSP>

RSP transport(const REQ &);

TRANSPORT LAYER

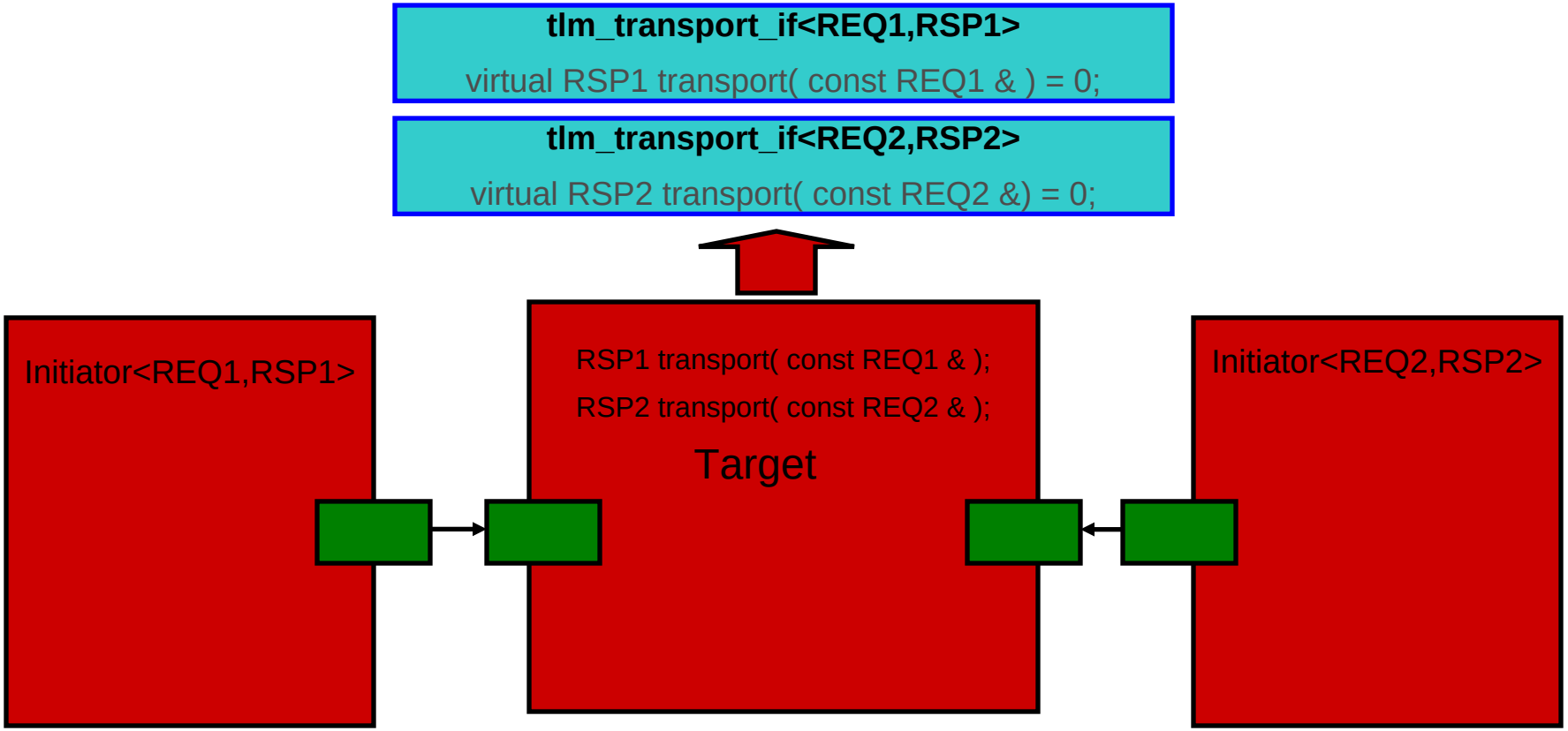
tlm_transaction

sc_ export

tlm_transport_if<REQ,RSP>

virtual RSP transport(const REQ &) = 0;

TLM transport : Advantages of templates – a different transport function for each protocol



Port Declarations

tlm_export<REQ1, RSP1> first(“port 1”);

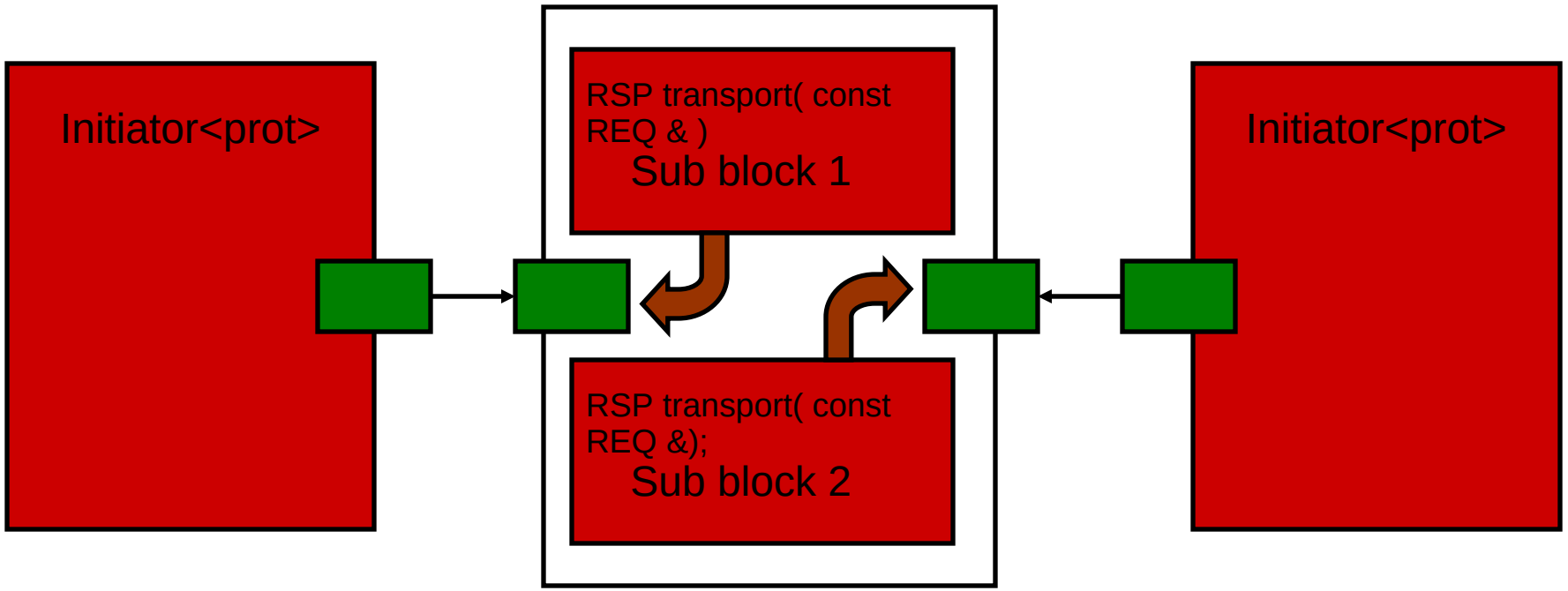
tlm_export<REQ2,RSP2> second(“port 2”);

Bindings in the constructor

first(this);

second(this);

TLM transport : Advantages of sc_export – multiple implementations of the same protocol



Port Declarations

tlm_export<REQ,RSP> first(“port 1”);

tlm_export<REQ,RSP> second(“port 2”);

Bindings in the constructor

first(block1);

second(block2);

Unidirectional API

- sc_fifo, or more precisely the sc_fifo interfaces, are the basic foundation of the unidirectional TLM API
  - sc_fifo provides unidirectional information transfer of any data type
  - Supports both non-blocking and blocking interface methods
  - Reliable communication mechanism, well understood how to model deterministic systems using sc_fifo
  - Safe to use in concurrent systems
  - Well understood how to perform static scheduling to optimize simulation and implementation performance
  - Users understand sc_fifo, so this will help them understand any TLM standard which is based on the sc_fifo interfaces
  - Varying the storage capacity of sc_fifo instances within TLM models is an intuitive and useful way to perform design exploration
  - sc_fifo interfaces are usable with channels other than sc_fifo, or directly in the target by using sc_export.

sc_fifo_ifs.h

Changes to sc_fifo agreed to in the LWG and only recently implemented in SystemC 2.1

```
template <class T>
class sc_fifo_nonblocking_in_if : virtual public sc_interface
{
public:
  virtual bool nb_read( T& ) = 0;
  virtual const sc_event& data_written_event() const = 0;
};
template <class T>
class sc_fifo_blocking_in_if : virtual public sc_interface
{
public:
  virtual void read( T& ) = 0;
  virtual T read() = 0;
};
template <class T>
class sc_fifo_in_if :
  public sc_fifo_nonblocking_in_if<T>,
  public sc_fifo_blocking_in_if<T>
{
  virtual int num_available() const = 0;
};
```

```
template <class T>
class sc_fifo_nonblocking_out_if : virtual public sc_interface
{
public:
    virtual bool nb_write( const T& ) = 0;
    virtual const sc_event& data_read_event() const = 0;
};
template <class T>
class sc_fifo_blocking_out_if
: virtual public sc_interface
{
public:
    virtual void write( const T& ) = 0;
};
template <class T>
class sc_fifo_out_if :
  public sc_fifo_nonblocking_out_if<T>,
  public sc_fifo_blocking_out_if<T>
{
    virtual int num_free() const = 0;
};
```

split interfaces into blocking and non blocking (OSCI speak)

Unidirectional tlm interfaces

```
template < typename T >
class tlm_blocking_get_if : public virtual sc_interface
{
public:
  virtual T get( tlm_tag<T> *t = 0 ) = 0;
  virtual void get( T &t ) { t = get(); }
};

template < typename T >
class tlm_nonblocking_get_if : public virtual sc_interface
{
public:
  virtual bool nb_get( T &t ) = 0;
  virtual bool nb_can_get( tlm_tag<T> *t = 0 ) const = 0;
  virtual const sc_event &ok_to_get( tlm_tag<T> *t = 0 ) const = 0;
};

template < typename T >
class tlm_get_if :
  public virtual tlm_blocking_get_if< T > ,
  public virtual tlm_nonblocking_get_if< T > {};
```

```
template < typename T >
class tlm_blocking_put_if : public virtual sc_interface
{
public:
  virtual void put( const T &t ) = 0;
};

template < typename T >
class tlm_nonblocking_put_if : public virtual sc_interface
{
public:
  virtual bool nb_put( const T &t ) = 0;
  virtual bool nb_can_put( tlm_tag<T> *t = 0 ) const = 0;
  virtual const sc_event &ok_to_put( tlm_tag<T> *t = 0 ) const = 0;
};

template < typename T >
class tlm_put_if :
  public virtual tlm_blocking_put_if< T > ,
  public virtual tlm_nonblocking_put_if< T > {};
```

- names changed to avoid confusion with bus read, bus write, bus data
- nb_can_get, nb_can_put replace num_available and num_free()
- Tags enable implementation of many interfaces of the same type but different template parameters in the same module.

tlm channels

cadence

- tlm_fifo<T>
  - Current implementation of tlm_fifo is almost identical to sc_fifo
  - Extensions under consideration :
    - peek / pop / poke
    - Shrink / unshrink
    - Dynamic resizing
  - Any extensions will use request/update and clearly distinguish between blocking and nonblocking.
- tlm_req_rsp_channel<req,rsp>
  - Two fifos, one for requests and one for responses
  - It also implements tlm_transport_if<req,rsp>
  - So it can act as converter between bidirectional and unidirectional modules eg for arbitration