TLM Events
Making Temporal Decoupling Work

Dr. Jakob Engblom
System Simulation Center (SSC)
Intel
Stockholm, Sweden

accellera SYSTEMS INITIATIVE

SYSTEMC™ EVOLUTION DAY
MAY 3, 2016 | MUNICH | GERMANY

Presentation Copyright Permission

– A non-exclusive, irrevocable, royalty-free copyright permission is granted by **Intel** to use this material in developing all future revisions and editions of the resulting draft and approved Accellera Systems Initiative **SystemC** standard, and in derivative works based on this standard.

Why TLM?

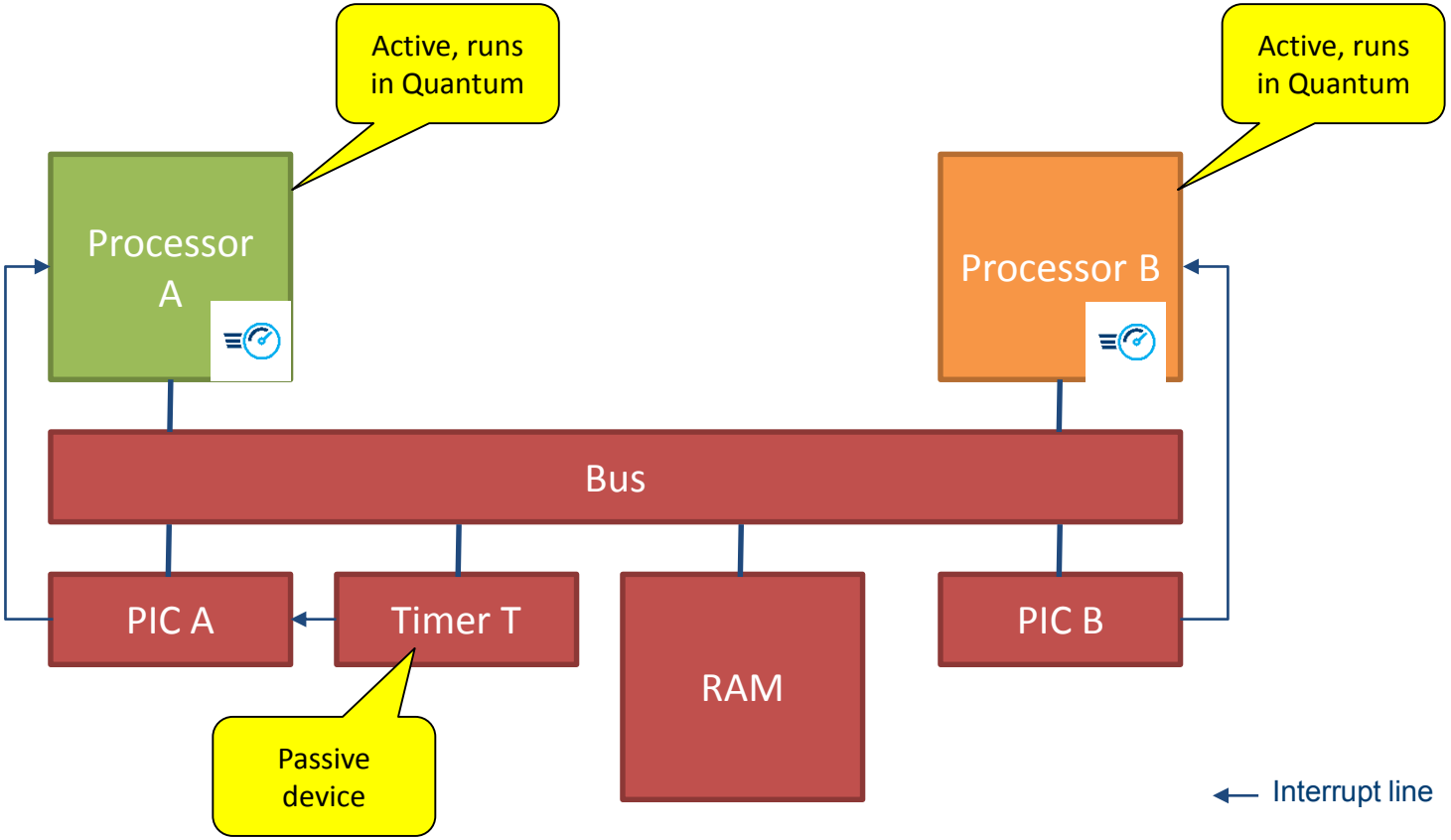
- Performance and ease and speed of modeling

- Communicate the what, not the how
  - Focus on the essential contents of the communication, not how the information is encoded or transported in the hardware
  - Implement what a device does, not how it does it in hardware
- Communication in a single step
  - Less scheduling per communication
  - Do not play out bus protocols
- Simpler models
  - No need to deal with protocols and cycles, just get a function call and do the work!
- Only consume processor resources when there is work to be done
  - TLM models are not clocked or polling; they are event-driven and reactive
- SystemC: Avoid getting the kernel involved
  - SC_Method + temporal decoupling
  - Topic of the day!
  - (Less of an issue in other simulation frameworks built with different semantics)

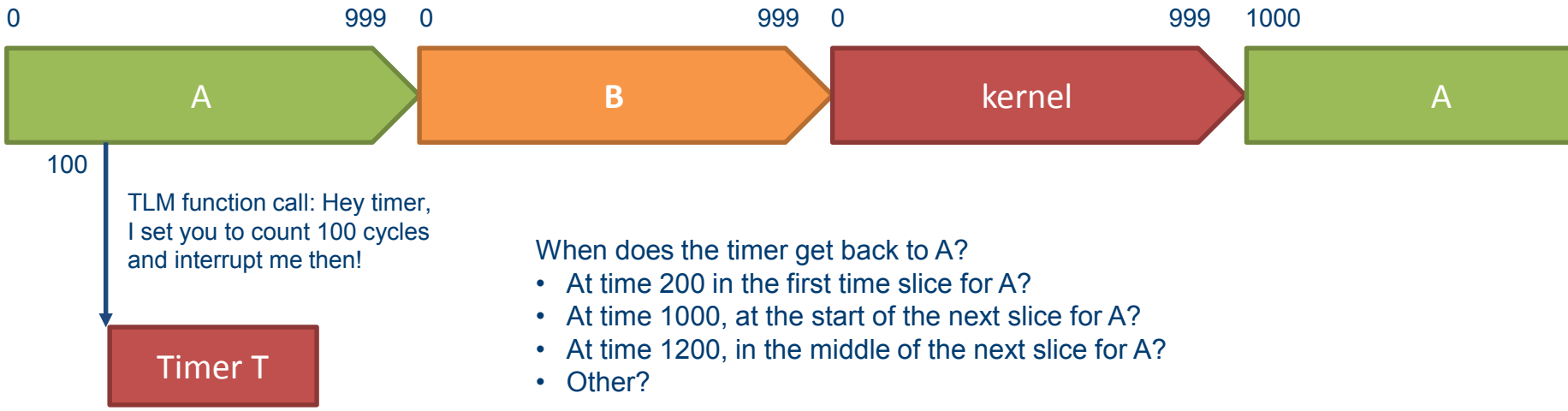
SystemC TLM-2.0 Temporal Decoupling: a bit Vague

- TLM-2.0 design:
  - Keep the single kernel time of traditional SystemC
  - Selected parts of simulation can be allowed to virtually run ahead (temporal decoupling), creating a “local” time
  - Implicitly, at the end of the quantum, run the kernel
    - This triggers queued kernel events, signals, etc.
    - Get global and “local” times in sync
- What happens when units communicate – and when?
  - Easiest solution: all communications happen at end of quantum, which is true for everything facilitated by the kernel
  - But TLM does not need the kernel, since TLM is a function call
  - Thus, TLM implies communication without involving the SystemC kernel – which is a large part of the point of TLM-SystemC

Example (Highly Simplified)



The Device Activation Time Issue



When does the timer get back to A?

- At time 200 in the first time slice for A?
- At time 1000, at the start of the next slice for A?
- At time 1200, in the middle of the next slice for A?
- Other?

This comes down to the question of what time domain Timer T is operating in.

- Is it in the kernel's domain?
- Should we associate it to a processor, such as A?
- **I think we need clearer rules for how to deal with this**

The Standard

”For example, consider the simulation of a system consisting of a processor, a memory, a timer, and some slow external peripherals. The software running on the processor spends most of its time fetching and executing instructions from system memory, and only interacts with the rest of the system when it is interrupted by the timer, say every 1 ms. The ISS that models the processor could be permitted to run ahead of SystemC simulation time with a quantum of up to 1 ms, making direct accesses to the memory model, ***but only synchronizing with the peripheral models at the rate of timer interrupts.***”

- I.e., sound like the timer will call back in the kernel’s time slice
- But this is just an example
- And the assumption that peripherals are ”slow” is not universally true

The Standard – LT Style

"A loosely-timed model can also benefit from explicit synchronization in order to guarantee deterministic execution, but a loosely-timed model is able to make progress even in the absence of explicit synchronization points (calls to wait), ***because each initiator will only run ahead as far as the end of the time quantum before yielding control****. A loosely-timed model can increase its timing accuracy by using* ***synchronization on-demand****, that is, yielding control to the scheduler before reaching the end of the time quantum."*

- This seems to say that we might end our quantum as soon as we do a device access, which sounds rather inefficient, and is not what happens in practice in most models I have seen
- **It is not clear if a TLM function call to another device is a "synchronization" in the language of this standard**

The Standard – Quantum Keeper

a) For maximum simulation speed, all initiators should use temporal decoupling, and the number of other runnable SystemC processes should be zero or minimized.

b) In an ideal scenario, the only runnable SystemC processes will belong to temporally decoupled initiators, and ***each process will run ahead to the end of its time quantum before yielding to the SystemC kernel****.*

c) A temporally decoupled initiator is not obliged to use a time quantum if communication with other processes is explicitly synchronized. ***Where a time quantum is used, it should be chosen to be less than the typical communication interval between initiators****; otherwise, important process interactions may be lost, and the model may be broken.*

- Point c puts a really low limit on the length of a quantum: in our example, it means that a quantum > 100 is disallowed. For maximum performance, our experience is that you need a quantum of 100k cycles or more.

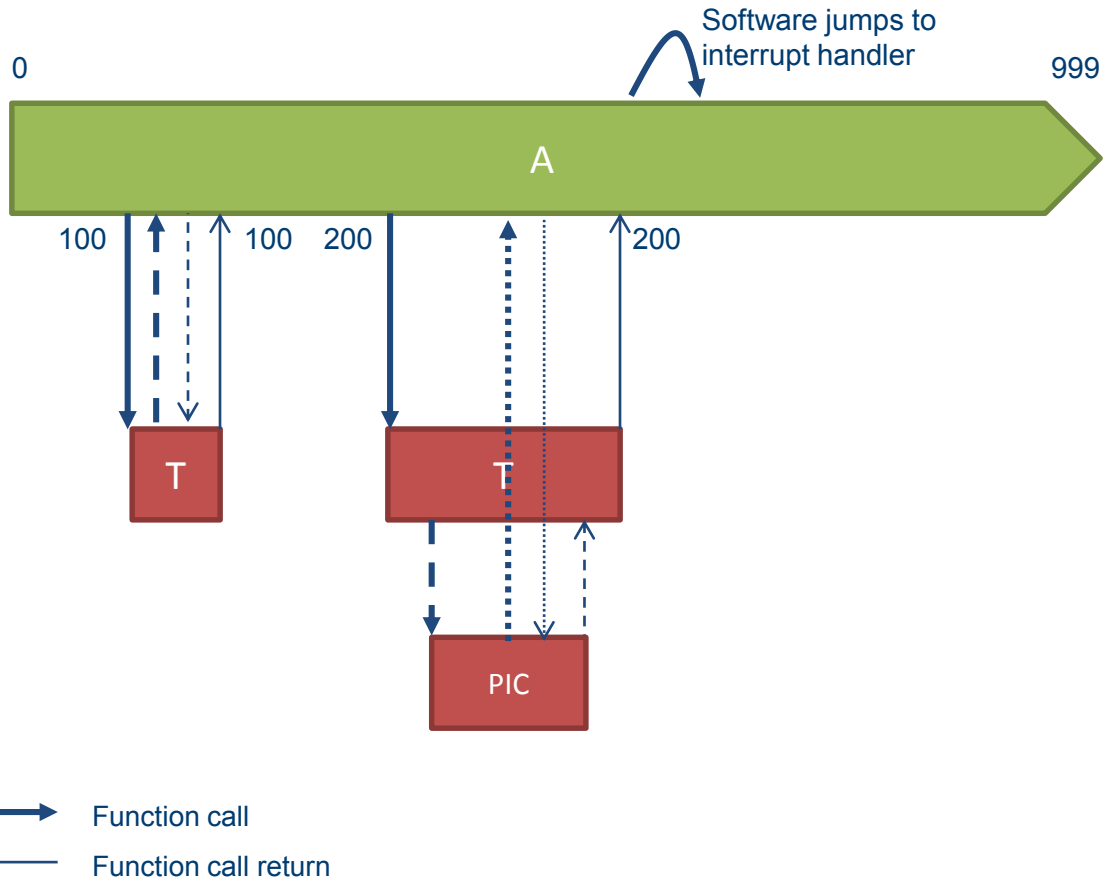
Time, Data, Interrupts

PROPOSAL: TLM EVENT

“TLM Events”: Time Management

- Allow *passive* device models (targets) to request callback from *active* objects (initiators)
  - Devices are not threads, but purely reactive (as they should be)
  - Model asks initiator to “please get me a callback in N cycles”
  - Initiator would be responsible for getting the callback to happen
  - As an adjunct, need to allow passive devices to query the time of the active object they work with
- Initiator can implement timed callbacks however it likes
  - *Manage its own queue* of timed events, and fire them off precisely (i.e., call back at time 200 in our example)
  - *Fall back to kernel*
  - We cannot reasonably standardize this aspect

TLM Events: Time: How it would work



(tA = 100) A to T: Hey timer, I set you to count 100 cycles and interrupt me then!

(tA = 100) T to A: Sure, here is an event that asks you to give me a callback 100 cycles from now

(tA = 100) T returns to A, finishing the TLM transaction in the device. Still at time 100.

(tA = 200) A calls T, since time is now 200. T gets a callback, and uses that to trigger an interrupt.

(tA = 200) T calls the PIC model over a **TLM interrupt or signal interface**

(tA = 200) PIC calls A over a TLM interrupt. Processor notes it has to take an interrupt following the current event.

Time: Delbergue et al DVCon 2016

- Proposes having initiators expose a ”Quantum Keeper API”
  - The name is a bit unfortunate, since it is a local time API, not an API to manage time quanta between active objects – but it is what TLM-2.0 says
- Makes sure targets can query time, etc, in addition to posting

Analysis of TLM-2.0 and it's Applicability to Non Memory Mapped Interfaces

Guillaume Delbergue
GreenSocs -
Bordeaux INP, CNRS IMS, UMR 5218
guillaume.delbergue@greensocs.com

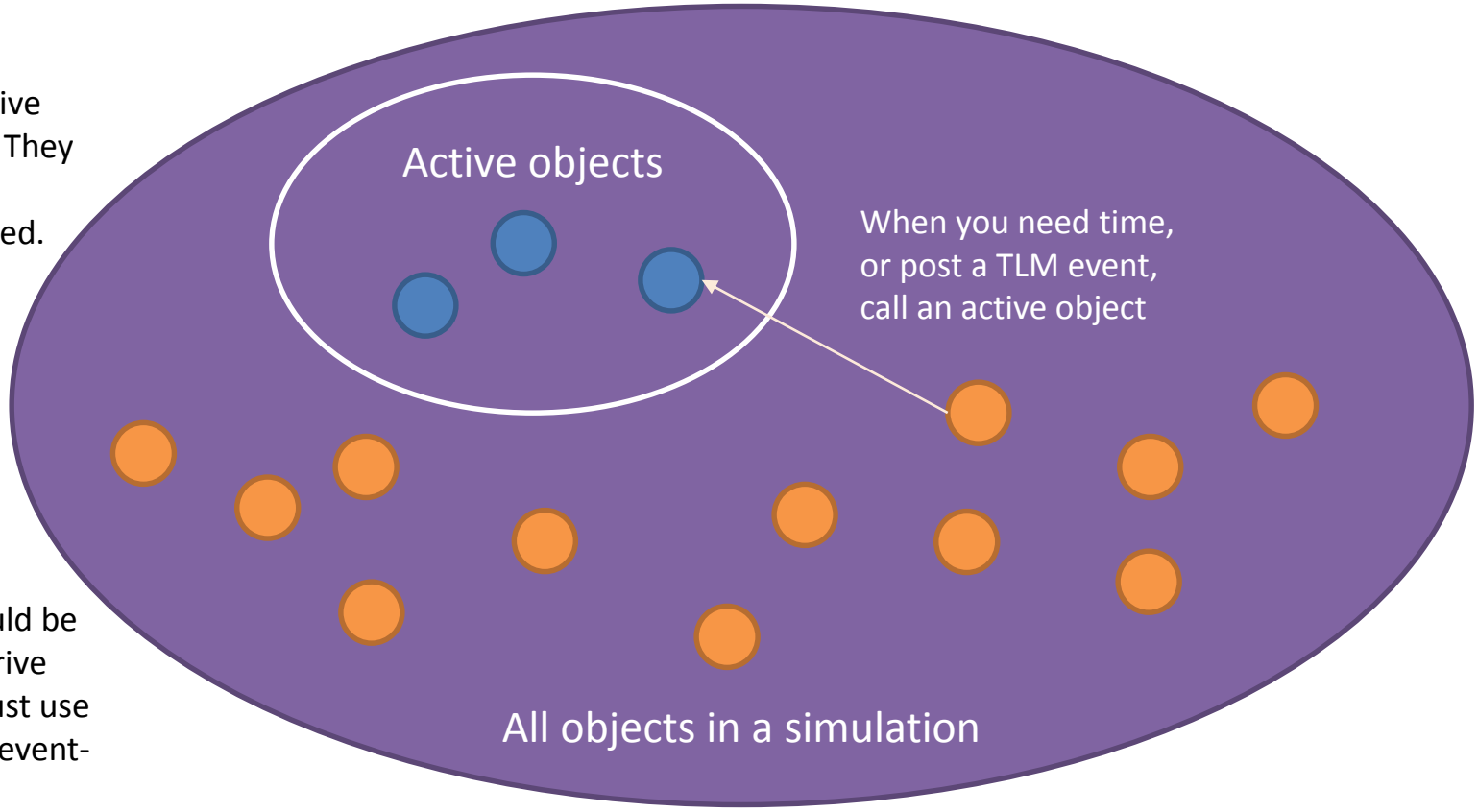
Mark Burton
GreenSocs
mark.burton@greensocs.com

Bertrand Le Gal and Christophe Jego
Bordeaux INP, CNRS IMS, UMR 5218
bertrand.legal@ims-bordeaux.fr
christophe.jego@ims-bordeaux.fr

***Abstract*—Virtual prototypes have become a critical tool for software development, architecture design, and importantly to bring software and hardware developments together. The introduction of the SystemC and the TLM-2.0 standard [1] has enabled a degree of interoperability and model re-use. However while the ambition for TLM-2.0 was wider, in reality it has only enabled interoperability for memory mapped bus based protocols. Since a typical virtual platform contains a wide variety of interconnect, individual designers have to define their own, often non-interoperable solutions. Our aim in this paper is to propose improvements to the existing TLM-2.0 standard, and a methodology around the TLM standard with the objective to facilitate the definition of a wide range of interfaces (not just memory mapped buses) in a consistent manner. Therefore in this paper, the existing Accellera TLM-2.0 standard is analyzed.**

TLM Events: Active and Passive Objects

Small subset of all objects will be active and provide time. They will be temporally decoupled for speed.



Most objects should be passive and not drive time, but rather just use TLM events to be event-driven.

TLM Events: Data

- Another aspect of event-driven modeling in TLM is that you often want to store data in events to know just what to do
  - Rather than registering tons of event types or having some kind of internal buffer memory in your device
- The TLM Event that a target posts to an initiator must carry a target-defined payload that will help the target make sense of the event when the callback comes
  - TLM-2.0 Payload Queue solves a similar problem, but in this case we have finished the memory operation so it is not applicable

Data: Haetzer & Radetzki FDL 2013

- Nice solution to extend current TLM events with data
- Something like this is needed in a TLM Event

SystemC Transaction Level Modeling with Transaction Events

Bastian Haetzer, Martin Radetzki

Embedded Systems Engineering Group
University Stuttgart
Pfaffenwaldring 5b
D-70569 Stuttgart, Germany
{haetzer, radetzki}@informatik.uni-stuttgart.de

TLM Events: Interrupts / Signals

- *This is a realization, post-submission, that I could not do events without talking interrupts/signals*
- As an adjunct to the TLM Events, **we absolutely need TLM interrupts or signals**
  - A way to call from one model to another *within* a time slice, without involving the kernel or ending the time slice
  - TLM-2.0 Memory-Mapped Bus is overkill, and also has response-reply semantics which is unnecessary
  - A unidirectional TLM signal is needed
    - Probably carrying data to allow modeling of signal buses or signals with data attached (not just a single line)
    - Might have bus/router/mux objects to implement many-to-one and one-to-many semantics

QUESTIONS?
COMMENTS?