May 2008

OSCI TLM-2 USER MANUAL

Software version: TLM-2.0 Final

Document version: DRAFT_JA11

Contributors

The TLM-2.0-final standard was created under the leadership of the following individuals:

Bart Vanthournout, CoWare, TLM-WG Chair
James Aldis, Texas Instruments, TLM-WG Vice-Chair

This document was authored by:

John Aynsley, Doulos

The following is a list of active technical participants in the OSCI TLM Working Group at the time of the release of TLM-2.0-final:

Tom Aernoudt, CoWare
James Aldis, Texas Instruments
John Aynsley, Doulos
Guillaume Audeon, ARM
Geoff Barrett, Broadcom
Bill Bunton, ESLX
Mark Burton, GreenSocs
Jerome Cornet, ST Microelectronics
Jack Donovan, ESLX
Jakob Engblom, Virtutech
Robert Guenzel, GreenSocs
Tim Kogel, CoWare
Laurent Maillet-Contoz, ST Microelectronics
Kiyoshi Makino, Mentor
Marcelo Montoreano, Synopsys
Victor Reyes, NXP
Olaf Scheufen, Synopsys
Bart Vanthournout, CoWare
Kaz Yoshinaga, Starc
Trevor Wieman, Intel
Charles Wilson, ESLX

The following is a list of active technical participants in the OSCI TLM Working Group at the time of the release of TLM-2.0-draft-2:

Tom Aernoudt, CoWare
James Aldis, OCP-IP
John Aynsley, Doulos
Guillaume Audeon, ARM
Bill Bunton, ESLX
Mark Burton, GreenSocs
Jack Donovan, ESLX
Othman Fathy, Mentor Graphics
Alan Fitch, Doulos
Karthick Gururaj, NXP
Atsushi Kasuya, Jeda
Tim Kogel, CoWare
Laurent Maillet-Contoz, ST Microelectronics
Marcelo Montoreano, Synopsys
Rishiyur Nikhil, Bluespec
Victor Reyes, NXP
Adam Rose, Mentor Graphics
Olaf Scheufen, Synopsys
Alan Su, Springsoft
Stuart Swan, Cadence
Bart Vanthournout, CoWare
Yossi Veller, Mentor Graphics
Trevor Wieman, Intel
Charles Wilson, ESLX

The following people have also contributed to the OSCI TLM Working Group:

Mike Andrews, Mentor Graphics
Matthew Ballance, Mentor Graphics
Geoff Barrett, Broadcom
Ryan Bedwell, Freescale
Bishnupriya Bhattacharya, Cadence
Bobby Bhattacharya, ARM
Axel Braun, University of Tuebingen
Herve Broquin, ST Microelectronics
Adam Erickson, Cadence
Frank Ghenassia, ST Microelectronics
Mark Glasser, Mentor Graphics
Andrew Goodrich, Forte Design
Serge Goosens, CoWare
Thorsten Groetker, Synopsys
Robert Guenzel, GreenSocs
Kamal Hashmi, SpiraTech
Holger Keding, Synopsys
Devon Kehoe, Mentor Graphics
Wolfgang Klingauf, GreenSocs
David Long, Doulos
Kiyoshi Makino, Mentor
Mike Meredith, Forte Design
David Pena, Cadence
Nizar Romdhane, ARM
Stefan Schmermbeck, Chipvision
Shiri Shem-Tov, Freescale
Jean-Philippe Strassen, ST Microelectronics
Tsutomu Takei, STARC
Jos Verhaegh, NXP
Maurizio Vitale, Philips Semiconductors
Vincent Viteau, Summit Design
Thomas Wilde, Infineon
Hiroyuki Yagi, STARC
Eugene Zhang, Jeda

Contents

1 Overview

This document is the User Manual for the OSCI Transaction Level Modeling standard, version 2.0-final. This version of the standard supersedes versions 2.0-draft-1 and 2.0-draft-2, and is not generally compatible with either. This version of the standard includes the core interfaces from TLM 1.0.

TLM-2.0 consists of a set of core interfaces, initiator and target sockets, the generic payload, utilities and analysis ports. The core interfaces include the core interfaces from TLM-1 together with the blocking and non-blocking transport interfaces, the direct memory interface (DMI), the debug transaction interface, the write interface, and the analysis interface. The core interfaces introduced in TLM-2 support loosely-timed and approximately-timed coding styles. The generic payload supports the abstract modeling of memory-mapped buses, together with an extension mechanism to support the modeling of specific bus protocols whilst maximizing interoperability. The quantum keeper and payload event queue (PEQ) are utility classes for use with the blocking and non-blocking transport interface.

The TLM-2 classes are layered on top of the SystemC class library as shown in the diagram below. The core interfaces can be used alone without the higher layers. The core interfaces and the initiator and target sockets can be used together without the generic payload. It is technically possible to use the generic payload directly with the core interfaces without using the initiator and target sockets, although this approach is not recommended. For maximum interoperability, it is recommended that the core TLM-2 interfaces, sockets and generic payload be used together in concert.

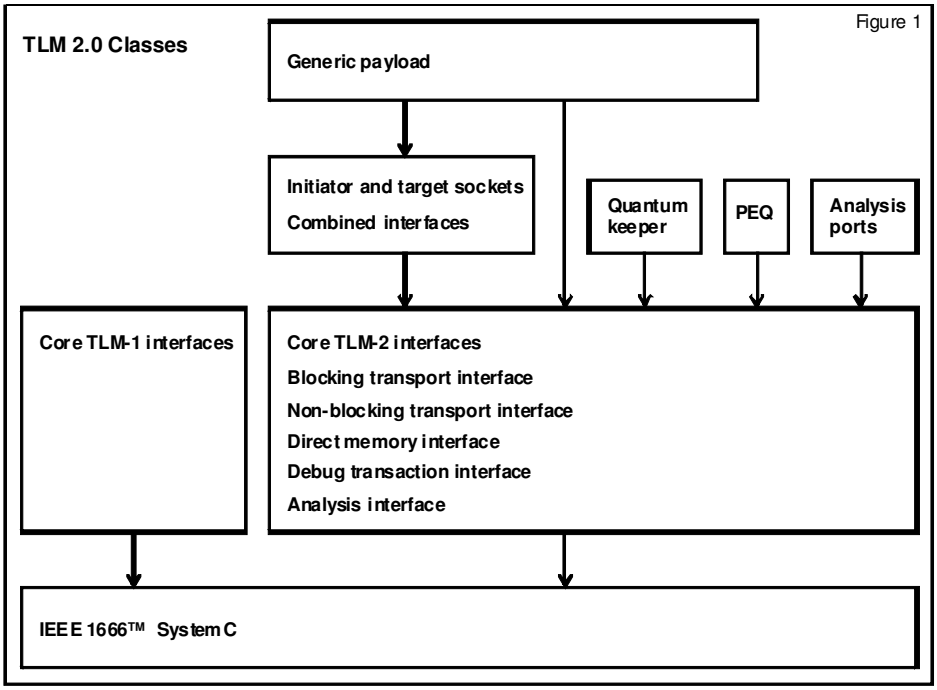


Figure 1

1.1 Scope

This document describes the contents of the TLM-2.0 standard. The main focus of this document is the key concepts and semantics of the TLM-2 core interfaces and classes. It does not describe all the supporting code, examples, and unit test. It lists the core TLM-1 interfaces, but does not define their semantics. This document is not a definitive language reference manual. It is the intention that this document will be extended over time to add more practical guidelines on how to use TLM-2.0

1.2 Source code and documentation

The TLM-2.0-final 2 release has a hierarchical directory structure as follows:

docs	Documentation, including User Manual, white papers, and Doxygen
examples	A set of application-oriented examples with their own documentation
include	The C++ source code, including readme files and release notes
unit_tests	A set of regression tests
utilities	A set of supplementary classes that are not part of the interoperability interface

The **docs** directory includes HTML documentation for the C++ source code created with Doxygen. This gives comprehensive text-based and graphical views of the code structured by class and by file. The entry point for this documentation is the file **docs/doxygen/html/index.html**.

2 References

This standard shall be used in conjunction with the following publications:

ISO/IEC 14882:2003, Programming Languages—C++

IEEE Std 1666-2005, SystemC Language Reference Manual

Requirements Specification for TLM 2.0, Version 1.1, September 16, 2007

2.1 Bibliography

The following books provide useful background information:

Transaction-Level Modeling with SystemC, TLM Concepts and Applications for Embedded Systems, edited by Frank Ghenassia, published by Springer 2005, ISBN 10 0 387-26232-6(HB), ISBN 13 978-0-387-26232-1(HB)

Integrated System-Level Modeling of Network-on-Chip enabled Multi-Processor Platforms, by Tim Kogel, Rainer Leupers, and Heinrich Meyr, published by Springer 2006, ISBN 10 1-4020-4825-4(HB), ISBN 13 978-1-4020-4825-4(HB)

ESL Design and Verification, by Brian Bailey, Grant Martin and Andrew Piziali, published by Morgan Kaufmann/Elsevier 2007, ISBN 10 0 12 373551-3, ISBN 13 978 0 12 373551-5

3 Introduction

3.1 Background

The TLM-1 standard defined a set of core interfaces for transporting transactions by value or const reference. This set of interfaces is being used successfully in some applications, but has three shortcomings with respect to the modeling of memory-mapped busses and other on-chip communication networks:

a) TLM-1 has no standard transaction class, so each application has to create its own non-standard classes, resulting in very poor interoperability between models from different sources. TLM-2 addresses this shortcoming with the generic payload.

b) TLM-1 has no support for timing annotation, so no standard way of communicating timing information between models. TLM-1 models would typically implement delays by calling **wait**, which slows down simulation. TLM-2 addresses this shortcoming with the addition of timing annotation to the blocking and non-blocking transport interface.

c) The TLM-1 interfaces require all transaction objects and data to be passed by value or const reference, which slows down simulation. Some applications work around this restriction by embedded pointers in transaction objects, but this is non-standard and non-interoperable. TLM-2 addresses this shortcoming with transaction objects whose lifetime extends across several transport calls, supported by a new transport interface.

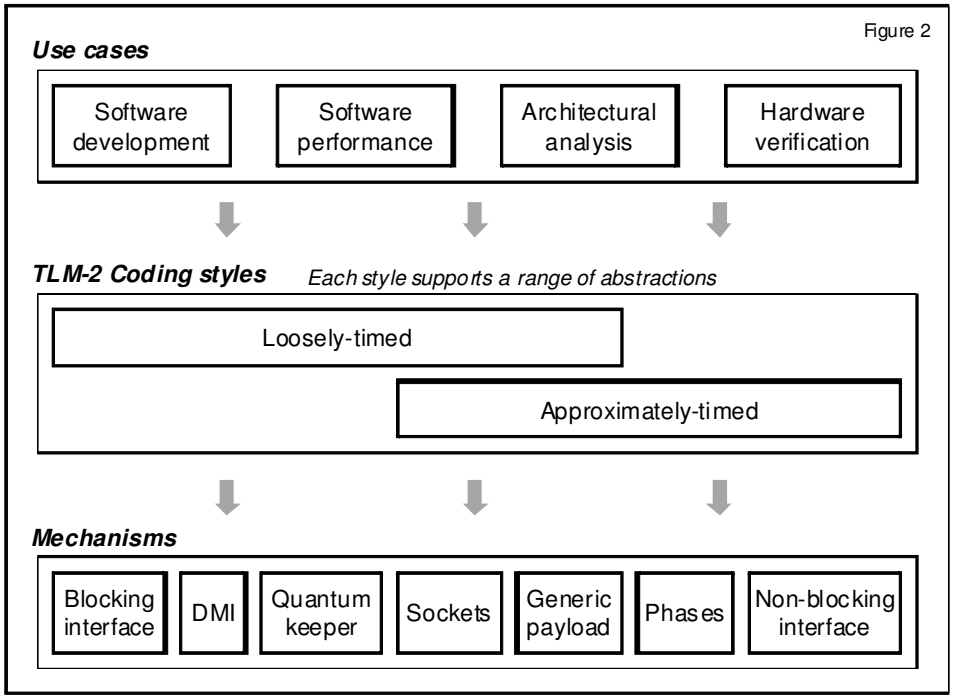
3.2 Transaction-level modeling, use cases and abstraction

There has been a longstanding discussion in the ESL community concerning what is the most appropriate taxonomy of abstraction levels for transaction level modeling. Models have been categorized according to a range of criteria, including granularity of time, frequency of model evaluation, functional abstraction, communication abstraction, and use cases. The TLM-2 activity explicitly recognizes the existence of a variety of use cases for transaction-level modeling (see the Requirements Specification for TLM-2.0), but rather than defining an abstraction level around each use case, TLM-2 takes the approach of distinguishing between interfaces (APIs) on the one hand, and coding styles on the other. The TLM-2 standard defines a set of interfaces which should be thought of as low-level programming mechanisms for implementing transaction-level models, then describes a number of coding styles that are appropriate for, but not locked to, the various use cases. Each coding style can support a range of abstraction across functionality, timing and communication.

An untimed functional model consisting of a single software thread can be written as a C function or as a single SystemC process, and is sometimes termed an *algorithmic* model. Such a model is not *transaction-level* per se, because by definition a transaction is an abstraction of communication, and a single-threaded model has no inter-process communication. A transaction-level model requires multiple SystemC processes to simulate concurrent execution and communication.

An abstract transaction-level model containing multiple processes (multiple software threads) requires some mechanism by which those threads can yield control to one another. This is because SystemC uses a co-operative multitasking model where an executing process cannot be pre-empted by any other process.

SystemC processes yield control by calling *wait* in the case of a thread process, or returning to the kernel in the case of a method process. Calls to *wait* are usually hidden behind a programming interface (API), which may model a particular abstract or concrete protocol that may or may not rely on timing information. Synchronization may be *strong* in the sense that the sequence of communication events is precisely determined in advance, or *weak* in the sense that the sequence of communication events is partially determined by the detailed timing of the individual processes. The former style is typified by the CSP (Communicating Sequential Processes) and KPN (Kahn Process Network) formalisms, and is easily implemented in SystemC using FIFOs, semaphores, or other synchronization primitives. This allows a completely untimed modeling style where in principle simulation can run without advancing simulation time, and all synchronization points are determined in advance and explicitly coded. The latter style is suitable when modeling multiple software threads running in an environment where the threads are not running in lockstep, but communicate from time-to-time, possibly at points in time that cannot be determined completely in advance. In this standard, such a coding style is called *loosely-timed*.



A more detailed transaction-level model may need to associate multiple protocol-specific timing points with each transaction, such as timing points to mark the start and the end of each phase of the protocol. By choosing an appropriate number of timing points, it is possible to model communication to a high degree of timing accuracy without the need to execute the component models on every single clock cycle. In this standard, such a coding style is called *approximately-timed*.

3.3 Coding styles

A coding style is a set of programming language idioms that work well together, not a specific abstraction level or software programming interface. TLM-2 recognizes several coding styles which should be used as a guide to model writing. For simplicity and clarity, this document restricts itself to discussing two specific named coding styles; *loosely-timed* and *approximately-timed*. In principle, it would be possible to define other coding styles based on the TLM-1 and TLM-2 mechanisms.

3.3.1 Untimed coding style

TLM-2 does not make explicit provision for an untimed coding style, because all contemporary bus-based systems require some notion of time in order to model software running on one or more embedded processors. However, untimed modeling is supported by the core TLM-1 interfaces.

3.3.2 Loosely-timed coding style and temporal decoupling

The loosely-timed coding style makes use of the blocking transport interface. This interface allows only two timing points to be associated with each transaction, corresponding to (but not necessarily simultaneous with) the call to and return from the blocking transport function. In the case of the generic payload, these two timing points mark the beginning of the request phase and the beginning of the response phase. In principle these two timing points could occur at the same simulation time, but more usually they would be skewed.

The loosely-timed coding style is appropriate for the use case of software development in an MPSoC environment, where multiple software threads are running on a virtual model of a hardware platform, possibly under the control of an operating system. The loosely-timed coding styles supports the modeling of timers and coarse-grained process scheduling, sufficient to model the booting and running of an operating system. It also supports temporal decoupling, where each software thread is permitted to run ahead in a local “time warp” until it reaches the point when it needs to synchronize with the rest of the system. Temporal decoupling can result in very fast simulation for certain systems.

The SystemC scheduler keeps a tight hold on simulation time. The scheduler advances simulation time to the time of the next event, then runs any processes due to run at that time or sensitive to that event. SystemC processes only run at the current simulation time (as obtained by calling the method **sc_time_stamp**), and whenever a SystemC process reads or writes a variable, it accesses the state of the variable as it would be at the current simulation time.

When a process finishes running it must pass control back to the simulation kernel. If the simulation model is written at a fine-grained level, then the overhead of event scheduling and process context switching becomes the dominant factor in simulation speed. One way to speed up simulation would be to allow processes to run ahead of the current simulation time, or to *warp* time.

In general, a process can be allowed to run ahead of simulation time until it encounters a dependency on a variable updated by another process, or needs to interact with another process. At that point, there is a decision to be made. In terms of general simulation techniques, outside the context of SystemC, there are several options. The process can be allowed to continue running, but the scheduler may need to backtrack later if it is discovered that the process assumed the wrong value. The scheduler may launch several alternative runs of the same process in parallel, assuming a different value for the external variable in each run, then pick the correct run after-the-fact when the value becomes known. This may make sense with

dedicated hardware support. A third option is to return control to the simulation kernel when an external dependency is encountered, only resuming the process later in simulation time when the external value becomes known.

In TLM-2, running ahead of simulation time is called *temporal decoupling*. Temporal decoupling does *not* permit backtracking, and each individual process is responsible for determining whether it can run ahead of simulation time without breaking the functionality of the model. When a process encounters an external dependency it has two choices: either force synchronization, which means yielding to allow all other processes to run as normal until simulation time catches up, or accept the current value and continue. The synchronization option guarantees functional congruency with the standard SystemC simulation semantics. Continuing with the current value relies on making a very significant assumption concerning communication and timing in the modeled system. It assumes that no damage will be done by sampling the value too early, and that any subsequent change to the value will be picked up in a subsequent process execution. This assumption is valid either if there is some explicit handshaking associated with the value, or if the value only changes relatively infrequently.

Temporal decoupling, and in particular allowing the time of process interactions to slip in time, is characteristic of the loosely-timed coding style.

If a process were permitted to run ahead of simulation time with no limit, the SystemC scheduler would be unable to operate and other processes would never get the chance to execute. This is avoided by the TLM-2 quantum keeper, which imposes an upper limit on the time a process is allowed to run ahead, known as the *quantum*. The quantum is set by the application, and the quantum value represents a tradeoff between simulation speed and accuracy. Too small a quantum forces processes to yield and synchronize very frequently, slowing down simulation. Too large a quantum introduces large timing inaccuracies, possibly to the point where important events are missed and the model ceases to function.

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 1ms. The ISS that models the processor could be permitted to run ahead with a quantum of up to 1ms, making direct accesses to the memory model, but only synchronizing with the peripheral models at the rate of timer interrupts. Depending on the detail of the models, this could give a simulation speed improvement of up to 1000X.

It is quite possible for some processes to be temporally decoupled and others not, and also for different processes to use different values for the time quantum. However, any process that is not temporally decoupled it likely to become a simulation speed bottleneck.

In TLM-2, temporal decoupling is supported by the **tlm_quantum_keeper** class and the timing annotation of the blocking and non-blocking transport interface.

3.3.3 Synchronization in loosely-timed models

An untimed model relies on the presence of explicit synchronization points in order to pass control between initiators at predetermined points during execution. 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 complete absence of explicit synchronization points, 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. Synchronization-on-demand in a loosely-timed model is equivalent to explicit synchronization in an untimed model.

3.3.4 Approximately-timed coding style

The approximately-timed coding style is supported by the non-blocking transport interface, which is appropriate for the use cases of architectural exploration and performance analysis. The non-blocking transport interface provides for timing annotation and for multiple phases and timing points during the lifetime of a transaction.

For approximately-timed modeling, a transaction is broken down into multiple phases, with an explicit timing point marking the transition between phases. In the case of the generic payload there are exactly four timing points marking the beginning and the end of the request and the beginning and the end of the response. Specific protocols may need to add further timing points, which may possibly cause the loss of direct compatibility with the generic payload.

It is possible to use the non-blocking transport interface for loosely-timed modeling by restricting the number of phases to two, although the blocking transport interface is generally preferred because of its simplicity.

The approximately-timed coding style cannot generally exploit temporal decoupling because of the need for timing accuracy. Instead, each process typically executes in lock step with the SystemC scheduler. Process interactions are annotated with specific delays. To create an approximately-timed model, it is generally sufficient to annotate two kinds of delay: the latency of the target, and the initiation interval or accept delay of the target. The annotated delays are implemented by making calls to the SystemC scheduler, that is, **wait**(delay) or **notify**(delay).

3.3.5 Characterization of loosely-timed and approximately-timed coding styles

The coding styles can be characterized in terms of timing points and temporal decoupling.

Loosely-timed. Each transaction has just two timing points, the start and the end. Simulation time is used, but processes may be temporally decoupled from simulation time. Each process keeps a tally of how far it has run ahead of simulation time, and may yield because it reaches an explicit synchronization point or because it has consumed its time quantum.

Approximately-timed. Each transaction has multiple timing points. Processes typically need to run in lock-step with SystemC simulation time. Delays annotated onto process interactions are implemented using timeouts (wait) or timed event notifications.

Untimed. The notion of simulation time is unnecessary Processes yield at explicit pre-determined synchronization points.

3.3.6 Switching between loosely-timed and approximately-timed modeling[1]

A model may switch from the loosely-timed to the approximately-timed coding style during simulation. The idea is to run rapidly through the reset and boot sequence at the loosely-timed level, then switch to approximately timed modeling for more detailed analysis once the simulation has reached an interesting stage.

3.3.7 Cycle-accurate modeling

Cycle-accurate modeling is beyond the scope of TLM-2 at present. It is possible to create cycle-accurate models using SystemC and TLM-1 as it stands, but the requirement for the standardization of a cycle-accurate coding style still remains an open issue, possibly to be addressed by a future OSCI standard.

In principle, the approximately-timed coding style can be extended to encompass cycle-accurate modeling by defining an appropriate set of phases together with rules concerning which attributes of the transaction can be modified and read in which phase, and by whom. In the limit, each phase represents a single cycle. The TLM-2.0 release includes sufficient machinery for this, but the details have not been worked out.

3.3.8 Blocking versus non-blocking transport interfaces

The blocking and non-blocking transport interfaces are distinct interfaces that exist in TLM-2 to support different levels of timing detail. The blocking transport interface is only able to model the start and end of a transaction, with the transaction being completed within a single function call. The non-blocking transport interface allows a transaction to be broken down into multiple timing points, and typically requires multiple function calls for a single transaction.

For interoperability, the blocking and non-blocking transport interfaces are combined into a single interface. A model that initiates transactions may used the blocking or non-blocking transport interfaces (or both) according to coding style. Any model that provides a TLM-2 transport interface is obliged to provide both the blocking and non-blocking forms for maximal interoperability, although such a model is not obliged to implement both interfaces internally.

TLM-2 provides a mechanism (the so-called *convenience socket*) to automatically convert incoming blocking or non-blocking transport calls to non-blocking or blocking transport calls, respectively. Converting transport call types does incur some cost, particularly converting an incoming non-blocking call to a blocking implementation. However, the cost overhead is mitigated by the fact that any approximately-timed model is likely to dominate simulation time anyway. The existence of even a single approximately-timed model is likely to wipe out the speed benefit to be gained from using exclusively loosely-timed models.

The strength of the blocking transport interface is that it allows a simplified coding style for models that are able to complete a transaction in a single function call. The strength of the non-blocking transport interface is that it supports the association of multiple timing points with a single transaction. In principle, either interface supports temporal decoupling, but the speed benefits of temporal decoupling are likely to be nullified by the presence of multiple timing points for approximately-timed models.

[1] The details of LT/AT switching are to be added before the final release of TLM2.0

3.3.9 Use cases and coding styles

The table below summarizes the mapping between use cases for transaction-level modeling and coding styles:

Use Case	**Coding style**
Software application development	Loosely-timed
Software performance analysis	Loosely-timed
Hardware architectural analysis	Loosely-timed or approximately-timed
Hardware performance verification	Approximately-timed or cycle-accurate
Hardware functional verification	Untimed (verification environment), loosely-timed or approximately-timed

