Formal Verification of a Pervasive Interconnect Bus System in a High-Performance Microprocessor

Thuyen Le[1] Tilman Glökler[1] Jason Baumgartner[2]

[1]IBM Deutschland Entwicklung GmbH, D-71032 Böblingen, Germany
[2]IBM Systems & Technology Group, Austin, TX 78758
{thuyenle|gloekler}@de.ibm.com baumgarj@us.ibm.com

Abstract

In our high-performance PowerPC processor, the correctness of the so-called pervasive interconnect bus system, which provides, among others, Test and Debug access via external interfaces like JTAG, is of utmost importance. In this paper, we describe our approach in formally verifying the correctness of this bus system to combat the coverage problem of simulation-based techniques. The bus system and the associated arbitration logic support several functionalities such as deadlock detection and resolution. In order to efficiently complete all of the required formal analysis for verification, we needed to leverage a variety of proof and semi-formal algorithms, as well as reduction and abstraction algorithms. Experimental results are provided to show the efficiency of this approach.*

1. Introduction

System-on-a-Chip (SoC) designs are intrinsically bus-centric designs. Processor cores, system peripherals and accelerators are often interconnected through standardized bus protocols, enabling scalability and modularity. In the design of high-performance processors, the bus-centric approach also applies to the functional layer, which consists of processor cores, caches, DMAs, memory controllers, I/O controllers, etc. Orthogonal to the functional layer, there is an additional so-called *pervasive layer* in high-performance processors. The pervasive layer provides the infrastructure for functionalities like Power-on-Reset, built-in self-test, debug and trace functionality, system monitoring of signals/registers at run-time, power management, etc. Pervasive logic is the underlying basis for many of the reliability, accessibility and serviceability features in the server processor world, without which it would be impossible to initialize, monitor, and debug the fabricated chip.

The bus-centric approach is also attractive for the implementation of the pervasive interconnect structure to reduce complexity. As this bus represents the backbone for Test and Debug access, its correctness is of utmost performance to the overall design. Bus correctness implies checks such as the proper routing of data from and to interconnected components, as well as arbitration correctness to ensure fairness in granting, and the absence of starvation and deadlock. In many ways, the correctness of the pervasive layer logic is even more critical than the correctness of the functional logic, since an error of the pervasive interconnect structure may well render a costly fabrication of a chip entirely unusable or un-testable, whereas a functional logic error at least enables the analysis of other aspects of the chip. Hence, applying formal verification techniques to verify the pervasive layer logic is a promising approach to combat the coverage problem of simulation-based techniques.

In this paper, we describe our experiences in formally verifying the correctness of an industrial pervasive interconnect bus system, which allows multiple internal and external masters to access on-chip logic blocks. The structure consists of two buses which are coupled by two unidirectional bridges. Both buses are multi-master capable and run at different asynchronous clock frequencies in different voltage domains. Masters and slaves are connected to the bus via common interface components that, together with the bus logic itself, represent our design-under-test (DUT). Existing and proven logic blocks connected to the DUT are replaced by behavioral models in the testbench to minimize the problem size. Our focus is on the bus correctness, especially to prove the absence of starvation and deadlock to which this design was particularly prone due to diverse aspects such as asynchronous clock domains and power management. Due to the complexity of the logic being verified, we needed to resort to a variety of formal algorithms

*Trademark of International Business Machines Corporation in the United States and/or other countries.

978-3-9810801-2-4/DATE07 © 2007 EDAA

to complete the proofs. We describe these algorithms in addition to the actual properties used to verify the system.

Related work. There have been publications that focus on the formal verification of bus-based protocol compliance.

[12] defines a set of protocol rules to verify an implementation of an AMBA bus and peripherals with AMBA-compliant interfaces. Their approach verifies the bus interface compliance of a specific unit such as master or slave. In contrast, we focus on verifying the inter-operation of several masters and slaves attached via the bus, comprising a greater degree of complexity than studied in prior work due to aggressive power management logic and the need for real-time deadlock resolution logic.

[11] presents a case study of formally specifying the AMBA bus protocol, and model checking that formal model to look for protocol errors. This is in contrast to our work where the actual interconnected design components are verified.

[5] discusses the formal verification of an IBM CoreConnect processor local bus. Similar to our approach, they use behavioral models of masters and slaves to verify an actual bus arbiter. In contrast to their work, ours encapsulates a larger portion of the implementation, including components of the actual master and slave logic necessary to encapsulate the entire bus interface. This larger design slice was necessary for verifying proper inter-operation of these components, and our verification covers all aspects of the bus functionality such as data routing and deadlock resolution logic, not merely arbitration correctness.

[4] proposed the usage of a hardware protocol kit to automatically generate formal testbenches from a formal documentation that is manually compiled for specific protocols. In contrast to our work, since dealing with a generic framework, they do not discuss the application of formal verification to any particular hardware implementation.

Outline. This paper is organized as follows. In Section 2 we discuss the design under test. Section 3 describes the setup of our formal verification testbenches and the formal verification toolset used in our verification process. In Section 4, we explain the properties used to verify our design under test, and the results of the verification task including chosen algorithms and bugs found. Finally, Section 5 concludes this work.

2 Pervasive Interconnect Bus

In a high-performance microprocessor, there is a need to access logic peripherals for a variety of purposes such as Test and Debug access, configuring the Power-On-Reset (POR) engine, run-time system monitoring, among others. Figure 1 depicts the pervasive interconnect bus structure that is implemented in our industrial chip project to arbitrate competing requests of several masters to shared slaves. Accesses can either be initiated by the internal masters such as the processor core(s) or by external masters via chip interfaces such as JTAG, and/or other proprietary protocols. To be more specific, the JTAG interface is attached to Bus 1 as master A, so that an external tester can access via master A on-chip logic resources such as processor cores represented as slave V.

All masters and slaves are connected to the bus via a common Master Interface (MIF) or Slave Interface (SIF) component, respectively. The use of common interfaces is advantageous, since it reduces design and verification overhead through enabling logic reuse. The logic in the unshaded region are existing and pre-verified IPs (Intellectual Properties) whereas the shaded region denotes the newly implemented logic that represents our DUT for formal verification. Note that for the sake of simplicity, only one master instance *A* and one slave instance *U* connected to Bus 1 are shown in addition to the master bridge *B*12 and slave bridge *B*21. In reality, Bus 1 has four master instances and six slave instances attached to it.

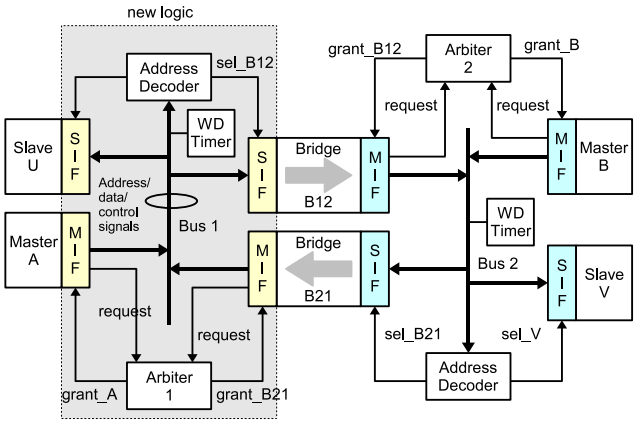

Figure 1. Pervasive Interconnect Bus

In Figure 1, the arrows show the direction of access initiation which is equivalent to the address flow whereas data flows in either direction, depending on whether a *read* or *write* access is performed. The two different buses run at different clock speeds and are connected to each other by two uni-directional bridges. Each bridge not only synchronizes the data across the asynchronous clock boundary, performs voltage level conversion, but also translates the communication protocol of Bus 1 to the protocol used by Bus 2. Both bus protocols are proprietary, employing a *single-envelope* protocol – i.e., once a transaction has been initiated, the transfer must be completed before another transaction can proceed over the bus.

Each bus provides a set of address and data signals as well as dedicated control signals. Only peripherals with master capability are able to initiate a transfer by raising their *request* signal. Write data to slaves and read data to masters are multiplexed onto the bus depending on the current granted access. The bus interconnect logic consists mainly of a *bus controller*, an *arbiter* and an *address decoder* (see Figure 2). The bus controller is the main finite state machine that, once a request has been detected, activates the arbiter, the address decoder and all other logic timely according to the phases as defined by the bus protocol. The arbiter selects a winning master among the pending incoming requests. The most-significant address bits of the winning master are then passed to the decoder to determine the target slave. A specific slave is selected by asserting its corresponding *sel* signal. After granting a master to access a target slave, the bus controller sets the internal logic such that the granted master address and master write data (for write access only) are available on the separated address and write data bus going to all SIFs attached to this bus, respectively. In case of a read access, the read data from the target slave are routed through the bus logic to the read data bus that is connected to all MIFs (see Figure 2). Parity generation or check is performed on address and data for accesses coming from and going over the bridge to protect data integrity from noise caused by voltage level translators.

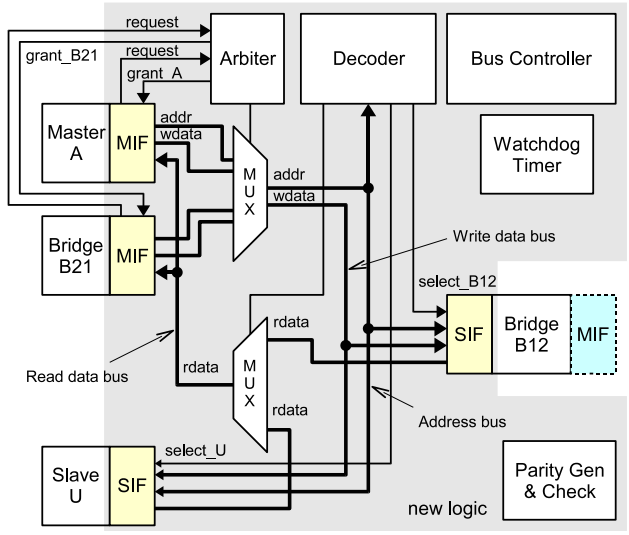

Figure 2. More details of Bus 1 with simplified dataflow of address, write and read data

Due to the defined bus structure as shown in Figure 1, the system may enter a starvation state in certain scenarios. For example, consider the case where Master A requests the bus to read from slave V, which is attached to Bus 2. Bus 1 arbiter thus grants the bus to Master A, since it is the only requesting master at this time. Bridge B12 is selected, which then transfers the request over the asynchronous clock boundary, performing the necessary bus protocol conversion. The master interface of B12 raises request to Bus 2. However, that bus may have been granted in the meantime to Master B. In this specific case, Master B wants to read from slave U, which is attached to Bus 1, but can not do so since Bus 1 is reserved for Master A. The transferred access of Master B over Bridge B21 will be blocked at Bus 1 as well as Bridge B12 at Bus 2. Since both buses are of single-envelope type, dedicated logic is necessary in the system to resolve this type of starvation, increasing the complexity of the implementation and verification.

Another scenario for starvation is the independent operation of the two clock domains. Due to aggressive power management, one clock domain can be shut off, possibly leaving the request signal of an attached master tied to logical one. This could translate to a constantly asserted bus request through the bridge to the other bus. The request will be served by the other bus, but the bridge can never respond according to the protocol, either leading to a starvation or a time-out that needs to be resolved. Figure 1 and 2 depict a watchdog timer to resolve the bus hang condition as just described.

The watchdog timer could also be used to detect the bus deadlock scenario described above and to time out after a certain number of cycles. This prior art solution, however, just postpones the deadlock scenario and hopes that it is unlikely that Master A and B will retry again within the critical time window that leads to the deadlock. In contrast to that, our Bus 1 implements a new mechanism that resolves the deadlock directly by *stealing silently* the bus from one master, and by handing it over to the other master. This new mechanism is one main target that needs to be verified using the formal verification approach described in the following sections. We stress here again that we are not only interested in formally verifying the arbiter logic but all the logic involved in the new bus system, i.e. arbiter, address decoder, datapath logic, watchdog timer, parity generation and checking as well as MIFs and SIFs.

3 Formal Verification Methodology

This section describes our formal verification approach as well as our selected tools and algorithms. The verification paradigm we adopt in this paper is that of a *testbench*, wherein one develops a set of *checkers* to assess the correct behavior of the design, in addition to a *driver* (sometimes called an *environment*) to constrain the input stimuli to avoid spurious failures. The testbench also consists of *initialization data* indicating which latches are to be initialized to a 0, 1, or random state. The verification task thus

consists of trying to obtain a counterexample trace from a specified initial state to one which drives a logical *one* onto each checker signal (which we refer to as *hitting* the check), or proving that no such counterexample exists. Note that the usage of the testbench within our formal verification context is to constrain the scenarios that the DUT can be exposed to.

We develop the testbench entirely in an HDL-based language so that it could be reused between formal verification, simulation, and emulation frameworks. Due to the potential size and complexity of the design slice being verified, we had initially thought that a *semi-formal* approach may be all that could be offered by our formal toolset, motivating the desire for a reusable specification. However, we ultimately were able to prove all of our properties, hence did not need to resort to simulation-based analysis.

In order to efficiently complete all of the required formal analysis, we needed to leverage a variety of proof and semi-formal algorithms, as well as a variety of reduction and abstraction algorithms dictated by the engines. The engines used in our verification process (please refer also to Table 2) include the following:

- **COM**: a combinational optimization engine, which attempts to merge functionally equivalent gates and rewrite logic cones to reduce their overall size [7, 8].
- **EQV**: a sequential redundancy removal engine, which first attempts to *guess* the set of gates that are functionally redundant across all time-frames, then a variety of algorithms including induction to prove that suspected redundancy [2, 9].
- **RET**: a min-area retiming engine [6] that reduces the number of state elements in the netlist by shifting them across combinational gates.
- **CUT**: an input-reducing engine that replaces a portion of the netlist local to the primary inputs with a functionally-corresponding piece of logic [10, 1].
- **LOC**: a localization engine, which isolates a critical subset of the netlist local to the properties by replacing internal gates by inputs. [13].
- **RCH**: a BDD-based reachability engine.
- **MOD**: a structural state-folding engine used to abstract certain clocking and latching schemes [3].
- **SCH**: a semi-formal search engine iterating bounded model checking (leveraging a circuit-based SAT solver with redundancy removal, and structural rewriting algorithms, similar to [7]) with random simulation and various state-prioritization schemes.

4 Verification Testbench

The partition into DUT and remaining logic has been depicted in Figure 1. We note that the remaining components are existing IPs that have been verified by other means. Hence, they are behaviorally modeled by drivers in our testbench, conservatively capable of producing a superset of the behavior that the actual logic can produce. Each master is modeled as being able to nondeterministically initiate requests, which covers all possible timing windows between a complete absence of requests to back-to-back requests. Response times of slaves as well as other relevant parameters of the bus protocol were similarly modeled in a maximally nondeterministic way. Though such modeling may result in behavior that does not comply with the actual interface specification such as non-terminating requests, this serves to verify recovery aspects of the system such as starvation detection.

Our testbench non-deterministically initializes all state elements in the design, and asserts the *reset* signal, distributed across all devices in the DUT, for the first clock period. This reset mechanism forces the design into a well-defined functional state; the masters are disallowed from driving any requests during this phase. After the reset phase, the masters are allowed to begin their requests. This setup validates, that the system properly initializes itself with its own reset mechanism regardless of what state it may be in, and that it behaves correctly after the reset phase. Note that we utilize a single testbench to *implicitly* test the reset mechanism, by validating that the correctness properties hold after the reset phase. The properties comprised in our testbench include:

- **Coverage events:** These must be hit in order to ensure that our testbench is able to drive all meaningful scenarios. Examples are events checking for reachability of all arbiter state machine states, and events checking for coverage of interesting sequences of transactions (like several occurrences of any of our bridge transactions). We suppress checking the coverage goals during the reset phase in order to avoid reporting bogus hits of coverage events due to arbitrary initialization states.
- **Assertions:** These correctness checks must be disable during the reset phase. Assertions include the liveness and fairness checks as well as the checks for data and address integrity. Special consideration was needed in case of a clock domain that was stopped and for data/address integrity checks in case of the associated transactions that resulted in timeouts. The liveness and fairness checks also verify the functionality of the watchdog timer as well as the *bus stealing* functionality for deadlock prevention: both features result in a substantially longer maximum duration between the master request event and the completion of the bus transaction.

4.1 Arbiter Properties

The most basic arbiter verification correctness property is that of *liveness* which means that every request is eventually granted and completed. Due to the computational expense of checking such unbounded liveness, we opted to check a more conservative *bounded liveness* property: every request will terminate within a pre-defined time bound. This time-bound is a function of parameters like the number of clock periods that are needed to complete a request (once granted), and the maximum possible number of outstanding requests. We implemented this bounded liveness check using a counter for each master that reflected the number of clock periods between an assertion of its request and the termination of its transaction, and verifying that this counter always remains less than the allowable upper-bound. This property also needs to consider the *stop clock* condition, which can occur at any time driven by our testbench, that stops one of the buses in Figure 1 and thus otherwise could cause uninteresting violations of the property. We recall that the *stop clock* condition is to model the effect of power management that can shut off the two clock domains of Bus 1 and Bus 2 independently.

We also checked fairness properties of the arbiter, which implements a weighted round-robin (WRR) scheme to assign an individual weight to each master. The assigned weight impacts how often its associated master should receive grants under heavy-load situations. Note that the well-known non-weighted round robin (RR) scheme, which assigns the same weight to all masters, is merely a special case of WRR. Both WRR and RR schedule work functionally by assigning slots to the masters. In the arbitration phase, a search over the slots is performed. The first slot owner, who happens to be also requesting the bus, will get granted. The search starts from a current slot pointer and is incremented modulo the total number of slots (hence *round robin*).

Table 1. Some possible slot assignments for WRR compared to a RR

Slot index (*mod* 6)	0	1	2	3	4	5
(a) Slot owner	M_1	M_2	M_1	M_3	M_1	M_4
(b) Slot owner	M_1	M_1	M_1	M_2	M_3	M_4
RR (c) Slot owner	M_1	M_2	M_3	M_4	–	–

In our DUT case, there are $N = 4$ masters denoted by M_i, $i = \{1,2,3,4\}$. Due to our system requirements, master M_1 is specified with a weight of 3 whereas the other masters have only a weight of 1. Two possible slot assignments (a) and (b) for this WRR are exemplarily given in Table 1. Note that the total number of unique slots is 6 although there are only 4 masters. Under heavy load situations, it can be observed that with assignment (a), the worst-case waiting time are 2 slots for M_1 and 6 slots for the other masters. With assignment (b), M_1 has a higher worst-case waiting time of 4 slots, but could be in turn granted 3 times in row. The selection of an assignment for a specific WRR implementation depends solely on the system requirements. Assignment (c) for the RR scheme is shown as a special case of WRR for comparison.

We use a general scheme consisting of a counter for each master as fail property to verify the priority scheme according to a given slot assignment. Instead of counting the number of clock periods between a request and grant, we count the number of grants given to any master *other than* the one associated with the counter. The counter starts counting whenever the associated master raises its request, and is reset whenever that master wins the arbitration phase. Each master i has a maximal count C_i that should never be reached, otherwise the assertion is violated. C_i must be set to the worst-case waiting time given by the implemented assignment. Given the implemented assignment (a) in Table 1, $C_1 = 2$ for M_1, and $C_2 = C_3 = C_4 = 6$.

4.2 Datapath Properties

In addition to the liveness and fairness properties, the correctness of the DUT requires us to verify the integrity of all address and data transmissions of the bus. We differentiate between read and write operations as follows:

- Write operations require checking that the correct slave is selected based on the most significant bits of the address. Furthermore, the least significant bits of the address and the write data must be correctly routed to the same slave.
- Read operations require checking the same properties for the address as write operations, but additionally require checking, that the read data from the slave is correctly routed back to the master.

In order to verify these properties, we implemented a checker that contains an address decoder compliant with the bus specification. This checker monitors the interface signals between all masters and slaves. If a master is granted access to the bus, the checker decodes its address and verifies that the slave activated by the DUT is the correct one. An additional check for read and write data as well as address bits was implemented for each master. Our testbench is able to handle the individual latency between bus input and bus output for any combination of master and slave, which is caused by the asynchronous boundaries.

4.3 Verification Results

The total number of properties that we have used in our verification testbench is 145. Among these 145 properties we have 4 liveness and 4 fairness properties as well as 4 data and 4 address integrity properties. Other properties cover illegal state transitions of the internal logic and correctness

Size Metric	Initial	COM	EQV	MOD,COM	EQV	RET	COM	CUT,COM	EQV	LOC[1]	CUT	RCH
Inputs	1388	118	115	115	108	414	169	157	156	112	46	0
AND Gates	16427	3951	2322	2015	1689	1658	1552	1522	1436	1058	895	0
Registers	2055	619	451	341	304	260	259	259	247	133	133	0
Properties	145	6	4	4	3	3	3	3	3	1[1]	1	0
Resources	7915s /640 MB											

Table 2. Verification results for arbitration assertions. [1]LOC performs a case-split per target; the largest abstraction is shown. Resources are cumulative across all properties.

of the parity detection logic; the remaining properties are coverage goals. Table 2 shows the unsolved properties as well as the design complexity in terms of its primary inputs, AND gates and registers after applying the formal reduction and proof engines mentioned in the first line. It also indicates the accumulative resources in terms of runtime and memory consumption on a computer running at 2.33GHz with 2GB memory. The applied engines have been shortly described in Section 3.

The coverage goals turned out to be extremely helpful to find and resolve problems in our environment: several non-trivial problems of our drivers have been found mainly caused by the fact, that each master and slave are supposed to behave slightly different. For the bridge interfaces, we found a subtle bug, that resulted in a starvation condition of the bridge $B12$ after the first successful transaction. Consequently, we implemented sequence checkers to cover the occurrence of several interesting events and combinations of events to increase our confidence in the testbench.

For the DUT we found 11 different design bugs that can be classified as follows:

- forgotten reset mechanism for important state latches
- incorrect arbitration (fairness violation) due to wrong state transitions in the arbiter resulting in a starvation of the complete bus
- problems with recovering from the clock stop condition
- wrong/forgotten implementation of clock edge detection logic

5 Summary and Conclusion

We have been discussing the formal verification of our pervasive interconnect bus system, which provides, among others, Test and Debug access to our high-performance processor. Despite the high complexity of the DUT together with its comprehensive testbench, we could formally prove all critical liveness, fairness and data integrity properties. This success was enabled by leveraging a variety of proof and semi-formal algorithms as well as different reduction and abstraction algorithms. It is also worth mentioning, that all of the design bugs would have been fatal in the final chip and would have rendered it mostly unusable. The majority of the discovered design bugs are corner cases: a simulation-based approach would most likely have missed them or at least would have required an excessive amount of simulation cycles to hit them. Last, but no least, the verification effort in terms of man hours for writing the drivers and checkers in our methodology and the runtime for the formal analysis makes it very attractive to use formal verification for more and more complex designs.

References

[1] J. Baumgartner and H. Mony. Maximal input reduction of sequential netlists via synergistic reparameterization and localization strategies. In *CHARME*, Oct. 2005.

[2] P. Bjesse and K. Claessen. SAT-based verification without state space traversal. In *FMCAD*, November 2000.

[3] P. Bjesse and J. Kukula. Automatic generalized phase abstraction for formal verification. In *ICCAD*, Nov. 2005.

[4] S. Dellacherie. Bringing automation to the verification of SoC based designs. In *GSPx 2005*, Oct. 2005.

[5] A. Goel and W. R. Lee. Formal verification of an IBM coreconnect^{TM} processor local bus arbiter core. In *DAC 2000*, 2003.

[6] A. Kuehlmann and J. Baumgartner. Transformation-based verification using generalized retiming. In *CAV*, July 2001.

[7] A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai. Robust Boolean reasoning for equivalence checking and functional property verification. *IEEE Trans. CAD*, Dec. 2002.

[8] A. Mishchenko, S. Chatterjee, and R. Brayton. DAG-aware AIG rewriting: A fresh look at combinational logic synthesis. In *DAC*, July 2006.

[9] H. Mony, J. Baumgartner, V. Paruthi, and R. Kanzelman. Exploiting suspected redundancy without proving it. In *DAC*, June 2005.

[10] I.-H. Moon, H. H. Kwak, J. Kukula, T. Shiple, and C. Pixley. Simplifying circuits for formal verification using parametric representation. In *FMCAD*, Nov. 2002.

[11] A. Roychoudhury, T. Mitra, and S. R. Karri. Using formal techniques to debug the AMBA system-on-chip bus protocol. In *DATE*, March 2003.

[12] C. Sayer and J. Sonander. Formal verification of AMBA bus systems. In *Information Quaterly, Vol. 2, No. 3*, 2003.

[13] D. Wang. *SAT based Abstraction Refinement for Hardware Verification*. PhD thesis, Carnegie Mellon Univ., May 2003.