Formal Verification of Programs That Use MPI One-Sided Communication

Salman Pervez[1], Ganesh Gopalakrishnan[1], Robert M. Kirby[1], Rajeev Thakur[2], and William Gropp[2]

[1] School of Computing
University of Utah
Salt Lake City, UT 84112, USA

[2] Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, USA

Abstract. We used formal-verification methods based on model checking to analyze the correctness properties of one existing and two new distributed-locking algorithms implemented by using MPI's one-sided communication. Model checking exposed an overlooked correctness issue with the first algorithm, which had been developed by relying only on manual reasoning. Model checking helped confirm the basic correctness properties of the two new algorithms, while also identifying the remaining problems in them. Our experience is that MPI-based programming, especially the tricky and relatively poorly understood one-sided communication features, stand to gain immensely from model checking. Considering that many other areas of concurrent hardware and software design now routinely employ model checking, our experience confirms that the MPI community can benefit greatly from the use of formal verification.

1 Introduction

Concurrent protocols are notoriously hard to design and verify. Experience has shown that virtually all nontrivial protocol implementations contain bugs such as deadlocks, livelocks, and memory leaks, despite extensive care taken during design and testing. Most of these bugs are basic *design* errors due to "unexpected" (untested) concurrent behaviors. Therefore, it stands to reason that if finite-state models of these protocols are created and *exhaustively* analyzed for the desired formal properties, robust protocol implementations would result. The technology for such finite-state modeling, property description, and exhaustive analysis developed over the past three decades—known as *model checking* [2]—has been successfully applied to numerous software and hardware systems. Model checking is now an integral part of the Windows Device Driver Development Kit [1]. Virtually all cache-coherence protocols developed and deployed in modern microprocessors have been verified by using model checking. However, although

concurrency and concurrent-programming bugs in parallel scientific programming are similar to those in other areas, we find little evidence of model checking being applied to verify parallel scientific programs.

In this paper, we conduct case studies that show the promise of the application of model checking in the area of parallel scientific programming using MPI. In particular, we focus on MPI one-sided communication [10]. Being (relatively) recently introduced and implemented, MPI one-sided communication is insufficiently understood and documented. One-sided communication involves shared-memory concurrency, which is known to be inherently harder to reason about than the message-passing concurrency of traditional MPI. One-sided communication exacerbates verification complexity because it guarantees only weak ordering semantics with respect to loads and stores, which can freely reorder within a given synchronization epoch. This paper demonstrates that, by using model checking, bugs in MPI programs that use one-sided communication can be caught easily, while expending only modest amounts of human and computer time.

After presenting background on MPI one-sided communication in Section 2, we provide an overview of model checking in Section 3. We then describe the design of an existing distributed byte-range locking algorithm [17] and its formal verification through model checking (Section 4). Model checking helped uncover the serious problem of a potential deadlock, which the authors of the algorithm were unaware of. Model checking also found a more benign problem of extra (zero-byte) sends in the algorithm, which might lend itself to an implementation-dependent correction using `MPI_Iprobe` and posted receives. However, this problem may well turn into a memory leak. We then present two other designs of the same algorithm, formally verify them using model checking, and provide empirical observations to interpret these model-checking results (Section 5). In Section 6, we conclude with a discussion of future work.

To our knowledge, nobody has applied model checking to analyze programs that use MPI one-sided communication. Siegel and Avrunin have used model checking to verify programs that use basic MPI point-to-point communication [13, 14]. Kranzlmüller used a formal event-graph based method to help understand MPI program executions [6]. Matlin et al. used the SPIN model checker to verify parts of the MPD process manager used in MPICH2 [9].

2 MPI One-Sided Communication

For lack of space, we review only the features of MPI one-sided communication relevant to this paper. One feature in MPI one-sided communication allows processes to gain exclusive access to communication windows in a block of code bracketed by `MPI_Win_lock` and `MPI_Win_unlock` calls [10]. Read and write accesses can be performed by a process holding exclusive access to a window through `MPI_Put` and `MPI_Get`. The main semantic difficulty stems from these put and get calls being not required to obey their syntactic program order in terms of when they are performed. It is well known (see, e.g., [15]) that such

ordering guarantees are crucial to the correctness of even simple concurrent protocols such as Peterson's mutual exclusion. The specification of one-sided communication in MPI further exacerbates the issue by introducing a complex set of informally stated rules that can easily lead to contradictory interpretations.[3] Common mistakes users make include nesting synchronization epochs on the same window object (such as a win_lock/unlock within a fence), doing read-modify-writes via a get-modify-put in the same synchronization epoch (even though gets and puts are defined to be nonblocking), and doing a put and a get to/from the same memory location in the same synchronization epoch. For example, the broadcast algorithms in Appendix B and C of [8] are incorrect because they rely on `MPI_Get` being a blocking function, which it is not. In implementations that take advantage of the nonblocking nature of `MPI_Get`, such as MPICH2 [16], the code will, indeed, go into an infinite loop. Since MPI one-sided communication can be implemented in a variety of ways [4], the result of making such mistakes is often implementation dependent: the program may work fine on some implementations and not on others.

3 Model Checking

Model checking is a term that has acquired an overloaded meaning. It essentially is the activity of exhaustively examining all possible behaviors of a *model* of a concurrent program (akin to wind-tunnel testing of scale models of airplanes). We consider *finite-state* model checking where the model of the concurrent system is expressed in a modeling language—Promela [5] in our case (all the pseudocodes expressed in this paper have an almost direct Promela encoding once the MPI constructs have been accurately modelled). By exhaustively executing the concurrent-system model, a model checker reveals its entire state-transition structure and is able to establish temporal properties, such as "always P" and "A implies eventually Q" with respect to this structure. The state graphs we generate are a result of the *interleaved* execution of various processes or threads. A fundamental problem with model checking is that reachable state graphs are exponential in the number of concurrent processes. The past three decades of research has, essentially, focused on getting a good handle on this exponential growth, so much so that astronomically large finite-state spaces—or often even many classes of infinite state spaces—can be handled by model checkers. Despite the very large state spaces of the MPI models discussed in this paper, our model checking runs finished within acceptable durations (often in minutes) on standard workstations.

4 Formal Verification of Byte-Range Locking

In [17], Thakur et al. present an algorithm implemented by using MPI one-sided communication (with passive-target lock-unlock synchronization) for coordinat-

[3] A collaborative project between the University of Utah and Argonne is addressing the issue of elucidating as well as formalizing this specification.

ing a collection of parallel processes contending for byte-range locks. We first describe the algorithm briefly, followed by a description of how we model checked it. Because of space limits, we cannot present the full pseudocode of the original algorithm; the reader may refer to the original paper [17] for details.

4.1 The Byte-Range Locking Algorithm

Each process keeps in a single common memory window (`lockwin`) its state consisting of a `flag` (initialized to 0) and the `start` and `end` values for the byte range (initialized to -1). A flag of 0 indicates that the process does not have the lock, while 1 indicates that it either has acquired the lock or wants to acquire the lock. A process updates its state and reads others' states by acquiring exclusive access to `lockwin` and making `MPI_Put` and `MPI_Get` calls. Since the processes acquire exclusive access, the actions of any one process on `lockwin` are guaranteed to be atomic with respect to the actions of other processes.

In order to acquire the lock, a process sets its `flag` to 1, updates its `start` and `end` values, and gets the corresponding values of other processes. It then checks whether any other process has set a conflicting byte range and has a flag value of 1. If it does not find such a process, it assumes that it has acquired the lock. Otherwise, it assumes that it does not have the lock, resets its flag to 0 via another lock-put-unlock, and blocks on a zero-byte `MPI_Recv` call, waiting for a process that has the lock to wake it up with a zero-byte send. The process will retry the lock after receiving the message. To release a lock, a process again acquires exclusive access, resets its flag to 0 and its start and end offsets to -1, and gets the values of other processes. If it finds a process with a conflicting byte-range (ignoring the flag), it sends a zero-byte message (via `MPI_SEND`) to wake up that process.

4.2 Checking the Byte-Range Locking Algorithm

To model the algorithm, we first needed to model the MPI one-sided communication constructs used in the algorithm and capture their semantics precisely as specified in the MPI Standard [10]. For example, the MPI Standard specifies that if a communication epoch is started with `MPI_Win_lock`, it must end with `MPI_Win_unlock` and that the put/get/accumulate calls made within this epoch are not guaranteed to complete before `MPI_Win_unlock` returns. Furthermore, there are no ordering guarantees of the puts/gets/accumulates within an epoch. Therefore, in order to obtain adequate execution-space coverage, *all permutations of put/get/accumulate calls in the epoch must be examined.* However, the byte-range locking algorithm uses the `MPI_LOCK_EXCLUSIVE` lock type, which means that while a certain process has entered the synchronization epoch, no other process may enter until that process has left. This makes the synchronization epoch an atomic block and renders all permutations of the calls within it equivalent from the perspective of other processes. Modeling the byte-range locking algorithm itself was relatively straightforward. (This experience augurs well for the checking of other algorithms that use MPI one-sided communication,

as one of the significant challenges in model checking lies in the ease of modeling constructs in the target domain using modeling primitives in the modeling language.) The complete Promela code used in our model checking can be found online [11].

When we model checked our model with three processes, our model checker, SPIN [5], discovered an error indicating an "invalid end state." Deeper probing revealed the following error scenario (explained through an example, which assumes that P1 tries to lock byte-range $\langle 1, 2\rangle$, P2 tries to lock $\langle 3, 4\rangle$, and P3 tries to lock $\langle 2, 3\rangle$):

- P1 and P3 successfully acquire their byte-range locks.
- P2 then tries to acquire its lock, notices conflict with respect to both P1 and P3, and blocks on the `MPI_Recv`.
- P1 and P3 release their locks, both notice conflicts with P2, and both perform an MPI_Send, when only one send is needed.

Hence, while P2 ends up successfully waking up and acquiring the lock, the extra `MPI_Send`s may accumulate in the system. This is a subtle error whose severity depends on the MPI implementation being used. Recall that the MPI Standard allows implementors to decide whether to block on an `MPI_Send` call. In practice, a zero-byte send will rarely block. Nonetheless, an implementation of the byte-range locking algorithm can address this problem by periodically calling `MPI_Iprobe` and matching any unexpected messages with `MPI_Recv`s.

We then modeled the system as if these extra `MPI_Send`s do not exhaust the system resources and hence do not cause processes to block. In this case, model checking detected a far more serious deadlock situation, summarized in Figure 1. P1 expresses its intent to acquire a lock in the range $\langle 10, 20\rangle$ (1), with P2 following suit (2). P1 acquires the lock (3), finishes using it and relinquishes it (4), and performs a send to unblock P2 (5). Before P2 gets a chance to change its global state, P1 tries to reacquire the lock (6). P1 reads P2's current flag value as 1, so it decides to block by carrying out events (10) and (12). At this point, P2 changes its global state, receives the message sent by P1 (8), and proceeds to reacquire the lock (9). P2 reads P1's current flag value as 1, so it decides to block by carrying out events (11) and (13). Both processes now block on receive calls, and the result is deadlock. We note that the authors of the algorithm were unaware of this problem until the model checker found it!

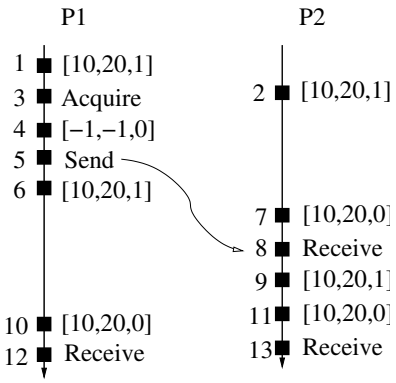

Fig. 1. A deadlock scenario found through model checking.

5 Correcting the Byte-Range Locking Algorithm

We propose two approaches to fixing this deadlock problem and describe our experience with using model checking on these solutions.

Alternative 1. One way to eliminate deadlocks is to add a third state to the "flag" used in the algorithm. This is shown in the pseudocode in Figure 2. In the original algorithm, a flag value of '0' indicates that the process does not have the lock, while a flag value of '1' indicates that it either has acquired the lock or is in the process of determining whether it has acquired the lock. In other words, the '1' state is overloaded. In the proposed fix, we add a third state of '2,' with '0' denoting the same as before, '1' now denoting that the process has acquired the lock, and '2' denoting that it is in the process of determining whether it has acquired the lock. There is no change to the lock-release algorithm, but the lock-acquire algorithm changes as follows.

When a process wants to acquire a lock, it writes its flag value as '2' and its start and end values in the memory window. It also reads the state of the other processes from the memory window. If it finds a process with a conflicting byte range and a flag value of '1,' it knows that it does not have the lock. So it resets its flag value to '0' and blocks on an `MPI_Recv`. If no such process (with conflicting byte range and flag=1) is found, but there is another process with a conflicting byte range and a flag value of '2,' the process resets its flag to '0,' its start and end offsets to -1, and retries the lock from scratch. If neither of these cases is true, the process sets its flag value to '1' and considers the lock acquired. An assessment of this protocol using model checking is presented later in this section.

Alternative 2. This approach uses the same values for the flag as the original algorithm, but when a process tries to acquire a lock and determines that it does not have the lock, it picks a process (that currently has the lock) to wake it up and then blocks on the receive. For this purpose, we add a fourth field (the pick field) to the values for each process in the memory window (see Figure 3). The process about to block must now decide whether to block. This decision is based on two factors: (i) Has the process selected to wake it up already released the lock? and (ii) Is there a possibility of a deadlock caused by a cycle of processes that wait on each other to wake them up? The latter can be detected and avoided by using the algorithm in Figure 4. The former can be easily determined by reading the values returned by the `MPI_Get` on line 26. If the selected process has already released the lock, a new process must be picked in its place. We simply traverse the list of conflicting processes until we find one that has not yet released the lock. If no such process is found, the algorithm tries to reacquire the lock. Note the added complexity of going through the list of conflicting processes and doing put and get operations each time. However, if this loop is successful and the process blocks on `MPI_Recv`, we can save considerable processor time in the case of highly contentious lock requests as compared with Alternative 1.

```
1   Lock_acquire (int start, int end)
2   {
3     val[0] = 2; /* flag */ val[1] = start; val[2] = end;
4     while (1) {
5       /* add self to locklist */
6       MPI_Win_lock(MPI_LOCK_EXCLUSIVE, homerank, 0, lockwin);
7       MPI_Put(&val, 3, MPI_INT, homerank, 3*myrank, 3, MPI_INT, lockwin);
8       MPI_Get(locklistcopy, 3*(nprocs-1), MPI_INT, homerank, 0, 1,
9               locktype1, lockwin);
10      MPI_Win_unlock(homerank, lockwin);
11      /* check to see if lock is already held */
12      conflict = flag1 = flag2 = 0;
13      for (i=0; i < (nprocs - 1); i ++) {
14        if ((flag == 1) && (byte ranges conflict with lock request))
15          { flag1 = 1; break; }
16        if ((flag == 2) && (byte ranges conflict with lock request))
17          { flag2 = 1; break; }
18      }
19      if (flag1 == 1) {
20        /* reset flag to 0, wait for notification, and then retry */
21        MPI_Win_lock(MPI_LOCK_EXCLUSIVE, homerank, 0, lockwin);
22        val[0] = 0;
23        MPI_Put(val, 1, MPI_INT, homerank, 3*myrank, 1, MPI_INT, lockwin);
24        MPI_Win_unlock(homerank, lockwin);
25        /* wait for notification from some other process */
26        MPI_Recv(NULL, 0, MPI_BYTE, MPI_ANY_SOURCE, WAKEUP, comm, &status);
27        /* retry the lock */
28        Lock_acquire(start, end); }
29      else if (flag2 == 1) {
30        /* reset flag to 0, start/end offsets to -1, and then retry */
31        MPI_Win_lock(MPI_LOCK_EXCLUSIVE, homerank, 0, lockwin);
32        val[0] = 0; /* flag */ val[1] = -1; val[2] = -1;
33        MPI_Put(val, 3, MPI_INT, homerank, 3*myrank, 3, MPI_INT, lockwin);
34        MPI_Win_unlock(homerank, lockwin);
35        /* retry the lock */
36        Lock_acquire(start, end); }
37      else {
38        MPI_Win_lock(MPI_LOCK_EXCLUSIVE, homerank, 0, lockwin);
39        val[0] = 1;
40        MPI_Put(val, 1, MPI_INT, homerank, 3*myrank, 1, MPI_INT, lockwin);
41        MPI_Win_unlock(homerank, lockwin);
42        /* lock is acquired */
43        break;
44      }
45    }
46  }
```

Fig. 2. Pseudocode for the deadlock-free byte-range locking algorithm (Alternative 1).

```
1   Lock_acquire (int start, int end)
2   {
3     int picklist[num_procs];
4     val[0] = 1; /* flag */ val[1] = start; val[2] = end;
5     val[3] = -1; /* pick */
6     while (1) {
7       /* add self to locklist */
8       MPI_Win_lock(MPI_LOCK_EXCLUSIVE, homerank, 0, lockwin);
9       MPI_Put(&val, 4, MPI_INT, homerank, 4*myrank, 4, MPI_INT, lockwin);
10      MPI_Get(locklistcopy, 4*(nprocs-1), MPI_INT, homerank, 0, 1,
11              locktype1, lockwin);
12      MPI_Win_unlock(homerank, lockwin);
13      /* check to see if lock is already held */
14      cprocs_i = 0;
15      for (i=0; i < (nprocs - 1); i ++)
16        if ((flag == 1) && (byte range conflicts with Pi's request)) {
17          conflict = 1; picklist[cprocs_i] = Pi; cprocs_i++; }
18      if (conflict == 1) {
19        for(j=0; j < cprocs_i; j++) {
20          MPI_Win_lock(MPI_LOCK_EXCLUSIVE, homerank, 0, lockwin);
21          val[0] = 0; val[3] = picklist[j];
22          /* reset flag to 0, indicate pick and pick_counter */
23          MPI_Put(&val, 4, MPI_INT, homerank, 4*myrank, 4, MPI_INT, lockwin);
24          MPI_Get(locklistcopy, 4*(nprocs-1), MPI_INT, homerank, 0, 1,
25                  locktype1, lockwin);
26          MPI_Win_unlock(homerank, lockwin);
27          if (picklist[j] has released the lock || detect_deadlock())
28            /* repeat for the next process in picklist */
29            j++;
30          else {
31            /* wait for notification from picklist[j] */
32            MPI_Recv (NULL, 0, MPI_BYTE, MPI_ANY_SOURCE, WAKEUP, comm,
33                      MPI_STATUS_IGNORE);
34            break; /* retry the lock */ }
35        }
36        /* if the entire list has been traversed, retry the lock */
37      }
38      else
39        break;  /* lock is acquired */
40    }
41  }
```

Fig. 3. Pseudocode for the deadlock-free byte-range locking algorithm (Alternative 2).

```
1   detect_deadlock() {
2     cur_pick = locklistcopy[4 * myrank + 3];
3     while(i < num_procs) {
4       /* picking this process means a cycle is completed */
5       if(locklistcopy[4 * cur_pick + 3] == my_rank) return 1;
6       /* no cycle can be formed  */
7       else if(locklistcopy[4 * cur_pick + 3] == -1) return 0;
8       else cur_pick = locklistcopy[4 * cur_pick + 3];
9     }
10  }
```

Fig. 4. Avoiding circular loops among processes picked to wake up other processes in Alternative 2.

Assessment of the Alternative Algorithms. We model checked these algorithms using SPIN, which helped establish the following formal properties of these algorithms:

- Absence of deadlocks (both alternatives).
- Communal progress (that is, if a collection of processes request a lock, then someone will eventually obtain it). Alternative 2 satisfies this under all fair schedules (all processes are scheduled to run infinitely often), whereas Alternative 1 places a few additional restrictions to rule out a few rare schedules (details in [12]).

We note that neither of these alternatives eliminates the extra sends, but, as described in Section 4, an implementation can deal with them by using `MPI_Iprobe`. That said, Alternative 2 considerably reduces these extra sends, as it restricts the number of processes that can wake up a particular process compared with Alternative 1. The exact performance tradeoffs of these algorithms will be determined as part of our future work. We are still seeking algorithms that avoid deadlock, avoid extra sends, and are efficient.

6 Conclusions and Future Work

We have shown how formal verification based on model checking can be used to find actual deadlocks in published algorithms that use the MPI one-sided communication primitives. We have also discussed how this technology can help shed light on a number of related issues such as forward progress and the possibility of there being unconsumed messages. We presented and analyzed two deadlock-free algorithms for byte-range locking and verified their characteristics.

Nonetheless, our work in this field is still in its early stages. Capitalizing on the maxim that formal methods can have their biggest impact when applied to constructs that are relatively new or are under development, we plan to formalize the entire set of MPI one-sided communication primitives. This can help develop a comprehensive approach to verifying programs that use the MPI one-sided constructs. As future case studies, we will analyze other algorithms, such as the scalable fetch-and-increment algorithm described in [3]. We plan to explore the use of automated tools to extract models from MPI programs, instead of creating them by hand. We also plan to explore the advantages of using other modeling languages, such as +CAL [7], and investigate the possibility of directly model checking MPI programs, instead of their extracted formal models.

Acknowledgments

This work was supported by NSF award CNS-0509379 and by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38.

References

1. Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft. In *Proceedings of IFM 04: Integrated Formal Methods*, pages 1–20. Springer, April 2004.
2. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.
3. William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.
4. William Gropp and Rajeev Thakur. An evaluation of implementation options for MPI one-sided communication. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 12th European PVM/MPI Users' Group Meeting*, pages 415–424. LNCS 3666, Springer, September 2005.
5. Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.
6. Dieter Kranzlmüller. *Event Graph Analysis for Debugging Massively Parallel Programs*. PhD thesis, John Kepler University Linz, Austria, September 2000. `http://www.gup.uni-linz.ac.at/~dk/thesis`.
7. Leslie Lamport. `http://research.microsoft.com/users/lamport/tla/pluscal.html`.
8. Glenn R. Luecke, Silvia Spanoyannis, and Marina Kraeva. The performance and scalability of SHMEM and MPI-2 one-sided routines on a SGI Origin 2000 and a Cray T3E-600. *Concurrency and Computation: Practice and Experience*, 16(10):1037–1060, 2004.
9. Olga Shumsky Matlin, Ewing Lusk, and William McCune. SPINning parallel systems software. In *Model Checking of Software: 9th International SPIN Workshop*, pages 213–220. LNCS 2318, Springer, 2002.
10. Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, July 1997. `http://www.mpi-forum.org/docs/docs.html`.
11. Salman Pervez. `http://www.cs.utah.edu/~spervez/model.tar.gz`.
12. Salman Pervez. Byte-range locks using MPI one-sided communication. Technical report, University of Utah, School of Computing, 2006. `http://www.cs.utah.edu/formal_verification/OnesidedTR1/`.
13. Stephen F. Siegel and George S. Avrunin. Verification of MPI-based software for scientific computation. In *Proceedings of the 11th International SPIN Workshop on Model Checking Software*, pages 286–303. LNCS 2989, Springer, April 2004.
14. Stephen F. Siegel, Anastasia Mironova, George S. Avrunin, and Lori A. Clarke. Using model checking with symbolic execution to verify parallel numerical programs. In *Proceedings of the ACM SIGSOFT 2006 International Symposium on Software Testing and Analysis*, July 2006.
15. Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, Inc., second edition, 2001.
16. Rajeev Thakur, William Gropp, and Brian Toonen. Optimizing the synchronization operations in MPI one-sided communication. *International Journal of High-Performance Computing Applications*, 19(2):119–128, Summer 2005.
17. Rajeev Thakur, Robert Ross, and Robert Latham. Implementing byte-range locks using MPI one-sided communication. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 12th European PVM/MPI Users' Group Meeting*, pages 120–129. LNCS 3666, Springer, September 2005.