Toward a Definition of and Linguistic Support for Partial Quiescence

Billy Yan-Kit Man[1], Hiu Ning (Angela) Chan[1], Andrew J. Gallagher[1], Appu S. Goundan[1], Aaron W. Keen[2], and Ronald A. Olsson[1]

[1]Department of Computer Science, University of California, Davis, Davis, CA 95616 USA
{many,chanhn,gallagha,goundan, olsson}@cs.ucdavis.edu

[2]Computer Science Department, California Polytechnic State University, San Luis Obispo, CA 93407 USA
akeen@csc.calpoly.edu

Abstract. The global quiescence of a distributed computation (or distributed termination detection) is an important problem. Some concurrent programming languages and systems provide global quiescence detection as a built-in feature so that programmers do not need to write special synchronization code to detect quiescence. This paper introduces *partial quiescence* (PQ), which generalizes quiescence detection to a specified part of a distributed computation. Partial quiescence is useful, for example, when two independent concurrent computations that both rely on global quiescence need to be combined into a single program. The paper describes how we have designed and implemented a PQ mechanism within an experimental version of the JR concurrent programming language. Our early results are promising qualitatively and quantitatively.

1 Introduction

In distributed programs, multiple processes cooperate to perform some task and communicate via messages to exchange information. One important, and well-studied, problem for such programs is to determine when the program's computation has completed, i.e., it has terminated normally or deadlocked. This *quiescence* problem is challenging because each process has only local information, but to solve the problem requires information about all processes (i.e., global state information). More formally, global quiescence (GQ) is defined as the state in which each process has terminated or deadlocked and there are no messages in the communication channels [14]. *Quiescence detection*, then, is the mechanism used to detect such a state in a distributed system.

Some programming languages and systems provide GQ detection as a built-in feature. That is, programmers do not need to write special synchronization code to detect quiescence. Instead, they can focus on writing application code. When quiescence is reached, the program can perform various actions such as simply terminating the program, outputting final results, gathering statistics from the overall computation, or initiating a new phase of the program, which might involve a new, corresponding phase of quiescence detection.

Although useful, GQ is limited to dealing with the state of *all* processes in a program. A more general, but more difficult to detect, property would define when a specified *part* of the program has become quiescent. For example, suppose we have two programs that use GQ and we want to combine them into a single program in which we want to perform different actions when each part of it becomes quiescent. This motivation led us to explore *partial quiescence* (PQ).

This paper proposes possible ways of defining PQ. It then discusses the particular definition selected and implemented in an experimental version of the JR concurrent programming language. (JR extends Java with a richer concurrency model [7, 18, 6].) The paper also shows how PQ leads to a different programming style for some problems. We compare the performance of using PQ detection and GQ detection. PQ might be a useful feature for other languages and libraries that define process or thread groups, as many do, and especially useful for those languages and libraries that already provide GQ detection.

Our work involves detecting quiescence (GQ or PQ) *dynamically*, i.e., during the actual execution of a concurrent program. An alternative approach involves *statically* determining various properties of concurrent programs, e.g., determining whether a concurrent program is deadlock-free. For example, [13] describes how to statically determine whether a given Ada program is deadlock-free and [9, 16, 17] describe how to statically verify that a program in notations such as process calculus with communication channels or timed automata with shared variables is *partial-deadlock* free. Partial deadlock means that a specified part of the program deadlocks. For example, [9] places restrictions on how processes communicate over channels and shows that the part of the program that uses "reliable" communication channels does not deadlock. The key difference between our approach and the static approaches, besides when the checking is performed, is that our approach treats quiescence as a normal part of program execution; the program itself is aware of its own quiescence and can react to quiescence as it desires.

The rest of this paper is organized as follows. Section 2 provides background on the general definition of GQ and how it has been incorporated as a built-in feature in some programming languages and systems; it describes how GQ is defined and implemented in JR and presents examples of programs that use GQ. Section 3 discusses the different ways of defining PQ. Section 4 discusses the definition of PQ we chose to provide in JR and gives examples of programs that use PQ. Section 5 presents an overview of our implementation and discusses its performance. Finally, Section 6 concludes. Further details appear in [12].

2 Background

2.1 Distributed Termination Detection (DTD)

As noted in Section 1, detecting the termination of a distributed computation is an important and challenging problem. A nice survey [14] describes DTD as follows. A distributed system consists of a collection of processessuch that processes communicate with each other by sending *activation messages* via some communication channels. An activation message is used not only for communication purposes among processes, but also for creation of a new process. A process is *active* if it is working on some computation or processing activation messages addressed to it. A process is *passive* if it is waiting for an activation message or termination. All processes in the system behave based on the following rules:

1. Activation messages can be generated only by active processes.
2. An active process may change its state to passive at any time.
3. A passive process may change its state to active only if it receives an activation message.

The above rules ensure that no further activation messages can be created in a system where all processes are passive: messages cannot be generated spontaneously. When the system has reached such a state, i.e., all processes are passive and no activation messages are in transit, then the system is *quiescent*. Quiescence detection is defined as the mechanism used to detect the state in which there are no messages in transit and all processes are waiting [14]. This definition generalizes that of DTD to both detecting termination as well as deadlock: i.e., sensing when the system is in a state from which it can no longer continue. Two main categories of DTD algorithms, as classified in [14], are wave algorithms (e.g., [3, 4, 19]) and credit distribution and recovery algorithms (e.g., [15]).

2.2 Tools and Systems with Support for Termination Detection

In some languages and systems (e.g., Ada, Java, MPI, and Pthreads), programs that reach a deadlock state wait indefinitely for the user to terminate them manually. However, in some cases, tools can assist in such detection. For example, Umpire [20] and MPI-CHECK 2.0 [11] detect deadlocks for MPI programs.

Some other programming languages or systems provide GQ detection as a built-in feature. GARLIC [8] extends Ada 95 with distributed programming features; it detects termination based on the algorithm proposed in [5]. JR [7, 18], SR [2, 1], and Charm [19] allow a quiescent program to output final results, gather statistics from the overall computation, or simply terminate the program. In this regard, JR and Charm are similar: when a program quiesces, it can initiate new computation, for which the quiescence feature can be used again. SR's quiescence feature is not as powerful: it is intended only for the program to clean up and terminate, and programs cannot use quiescence repeatedly.

Implementation of a Quiescence Feature in JR and SR. The implementation uses an approach that differs from the general DTD algorithms described in Section 2.1 because of their particular model of computation. A distributed program consists of a group of "virtual machines" (VMs). Each VM represents an address space, or unit of program distribution, and contains several processes, which can share variables within that address space or send message to other processes on that VM or to processes on other VMs. Typically, the number of VMs is not very large, but it varies as the program executes. The implementation uses a centralized manager to record information about all VMs in one place so as to make it easy to implement various services, such as an explicit exit (stop) from the program code, which needs to shut down all VMs. The implementation of DTD involves the RTS (run-time system) on each VM and the centralized manager. When a VM can make no further progress (i.e., all of its processes have terminated or are waiting to receive a message), it sends an idle message to the manager. This message contains the number of messages this VM has sent to each other VM and the number of messages this VM has received from each other VM. If the manager has received an idle message from each VM, it checks that no messages are in transit, specifically: for each VM VM_a, the number of sends from VM_a to each other VM, VM_b, matches the number of receives from VM_a reported by VM_b, If so, then the system is globally quiescent.

Example JR Program Using GQ. The program in Figures 1 and 2 (from [18]) performs matrix multiplication. Its `MMMain` class reads in two N × N matrices, instantiates a `MMMultiplier` object, and registers the operation `done` as the quiescence operation.[1] Its `MMMultiplier` class contains the processes that perform the actual computation. These processes begin execution after `MMMultiplier`'s constructor completes its execution. GQ is used to determine when these `compute` processes have finished their tasks. Once GQ has been detected, the registered operation `done` is invoked and its code outputs the resulting matrix. Without GQ detection, the programmer would need to write additional code to determine when the computation has terminated.

```
public class MMMain {
  private static MMMultiplier m;
  public static void main(String [] args) {
    double [][] A, B;  int N;  // A and B are NxN
    // read in NxN arrays A and B
    ...
    m = new MMMultiplier(A, B, N);
    // register done as the quiescence operation
    JR.registerQuiescenceAction(done);
  }
  private static op void done() { m.print(); }
}
```

Fig. 1. Matrix multiplication using GQ – `MMMain` class.

```
public class MMMultiplier {
  double [][] A, B, C;  int N; // A, B, and C are NxN
  public MMMultiplier(double [][] A, double [][] B, int N) {
    this.A = A; this.B = B; this.N = N;  C = new double [N][N];
  }
  process compute ( (int r = 0; r < N; r++), (int c = 0; c < N; c++) ) {
    // compute the inner product for C[r,c]
    C[r][c] = 0.0;  for (int k = 0; k < N; k++) { C[r][c] += A[r][k] * B[k][c]; }
  }
  public void print() { /* output C */  ...  }
}
```

Fig. 2. Matrix multiplication using GQ – `MMMultiplier` class.

[1] Technically, the registration needs to be within a `try/catch` block.

If no GQ operation is registered, then the program simply terminates when it quiesces. The quiescence operation can initiate new activity and can re-register the GQ operation (either the same or different operation), which will be invoked when the newly initiated activity quiesces.

3 Definition of Partial Quiescence (PQ)

Although GQ is useful, it restricts the detection to determine the quiescent state of *all* processes in a given program. Some notion of PQ, which addresses the quiescence of *part* of the program, would be useful. We want, for example, to combine two programs (i.e., two independent concurrent computations) that use GQ into a single program in which we want to perform different actions when each part of it becomes quiescent.

The first step is to define what PQ means. A natural approach is to apply quiescence to a group of processes in a program. Modifying the definition of quiescence from Section 2.1 to apply to a specific group of processes yields:

> Quiescence of group A is defined as the state in which (1) there are no messages in the system in transit to group A and (2) all processes in group A have terminated or are waiting for a message.

Because PQ deals with the interactions of groups of processes, it is, in general, more difficult to detect. This definition fits well if the process group is "closed" [10], i.e., only processes in group A send messages to processes in group A. However, this definition is not realistic if the process group is "open" [10], i.e., a message for a process in group A can be generated by a process outside of the group; such a message appears, from within group A, to have been generated "spontaneously". More concretely, a detection mechanism could detect that all processes in group A are passive and no message in transit is destined for group A, and so it would decide that group A is partially quiescent. However, that decision could be followed by a process outside group A sending a message to a process in group A. (In contrast, such spontaneous message generation is not possible for GQ (Section 2.1).)

A definition of PQ can deal with this spontaneous generation problem in various ways. One way would be to alter the above definition with a third clause, e.g., "and (3) no process outside of group A can possibly send to a process in group A". However, such a definition might not be useful: just because a process outside of group A can send a message to a process in group A does not guarantee that it ever actually will. Moreover, in general, keeping track of such information in a system where communication paths between processes is determined dynamically would be costly.

Therefore, we choose a weaker definition of partial quiescence, namely one that modifies (1) from the earlier definition:

> Quiescence of group A is defined as the state in which (1) there are no messages in the system from group A in transit to group A and (2) all processes in group A have terminated or are waiting for a message.

This definition fits well for closed process groups; the next sections illustrate that it is practical for open process groups.

4 JR Extended for Partial Quiescence

We have extended JR to support PQ. Now, JR programs can define groups of related processes and can register, for each process group, a *partial quiescence operation.* This section begins with examples to illustrate how PQ in the extended JR works and then discusses key aspects of the various mechanisms.

4.1 Expository Examples of PQ in JR

```
public class MMMain {
  private static MMMultiplier m1, m2;
  public static void main(String [] args) {
    double [][] A1, B1, A2, B2;  int N;  // A1, B1, A2, B2 are NxN
    // read in NxN arrays A1, B1, A2, B2
    ...
    ProcessGroup m_g1 = new ProcessGroup("Multiply Group1");
    ProcessGroup m_g2 = new ProcessGroup("Multiply Group2");
    JR.changeCreationGroup(m_g1); // processes within m1 will be in m_g1
    m1 = new MMMultiplier(A1, B1, N);
    JR.changeCreationGroup(m_g2); // processes within m2 will be in m_g2
    m2 = new MMMultiplier(A2, B2, N);
    // register partial quiescence operation for each process group
    JR.registerPartialQuiescenceAction(m_g1, done1);
    JR.registerPartialQuiescenceAction(m_g2, done2);
  }
  private static op void done1() { m1.print(); }
  private static op void done2() { m2.print(); }
}
```

Fig. 3. Multiple matrix multiplications using PQ – `MMMain` class.

Multiple Matrix Multiplications. The main program in Figure 3 shows how to use PQ to perform two simultaneous matrix multiplications. A nice attribute of our PQ approach is that the same `MMMultiplier` class from Figure 2 works here. The main program creates two process groups, one for each matrix multiplication. It uses `JR.changeCreationGroup` to specify the group in which newly created processes will be placed for each new matrix computation. (There is one default process group.) The main program then registers the PQ operation for each process group. When either group quiesces, its PQ operation will be invoked and that code outputs the results.

In contrast, consider a variant of the original main program in Figure 1 that starts two matrix multiplications and that uses GQ. It would wait for *both* computations to finish before outputting the result from either.

In Figure 3, two process groups might quiesce at about the same time, in which case the outputs from their quiescence operation might be interleaved. Their outputs can be serialized by deleting the present code for `done1` and `done2` (but keeping their `op` declarations) and adding the code in Figure 4 to the end of the `main` method. This code uses JR's multi-way receive statement (`inni`) to wait for an invocation of either of the PQ operations; it services one at a time, thus serializing their outputs.

```
  for (int i = 0; i < 2; i++) {
    inni void done1() {m1.print();}
    []   void done2() {m2.print();}
  }
```

Fig. 4. Code to serialize output from the multiple matrix multiplications.

Barrier Synchronization. PQ, as noted earlier for GQ, allows JR programs to re-register a quiescence operation. Consider the program in Figure 5 (from [18]). It shows a group of worker processes synchronizing their iterations via a barrier, implemented with semaphores.[2] The program also contains a coordinator process

```
public class Barrier {
  private static final int N = 10; // number of workers
  private static sem done = 0;
  private static cap void () proceed[] = new cap void()[N];
  static {  for (int i = 0; i < N ; i++) { proceed[i] = new sem; }  }
  private static process worker( (int i = 0; i < N; i++) ) {
    while (...) { // iterations remain
      // code to implement one iteration of task i
      ...
      // barrier
      V(done);       // tell coordinator "I did iteration i"
      P(proceed[i]); // wait for coordinator to say "continue"
    }
  }
  private static process coordinator {
    while (...) { // iterations remain
      for (int w = 0; w < N; w++) { P(done); }
      for (int w = 0; w < N; w++) { V(proceed[w]); }
    }
  }
  public static void main(String [] args) {
  }
}
```

Fig. 5. Barrier synchronization using semaphores.

that controls when workers begin their next iteration. The program uses an array of semaphores, `proceed` (one for each worker), rather than a single semaphore, to prevent a fast worker from "stealing" the message intended for a slow worker. With a single semaphore, a slow worker might be context switched after `V(done)` and before the `P(proceed)`, which would allow a fast worker to finish its iteration and get past the `P(proceed)`. (See [18] for details.)

This program can be rewritten using PQ and fewer semaphores, as shown in Figure 6. Worker processes no longer need to tell the coordinator that they are done (via the `done` semaphore); instead PQ will detect that. The role of the coordinator is no longer performed by a separate process. It is now the PQ operation that is invoked when all worker processes quiesce. Also, the `proceed` array of semaphores is now replaced with a single `proceed` semaphore: a fast worker cannot overtake a slow worker since all workers must quiesce before the coordinator operation is invoked and tells any worker it may proceed.

[2] In JR, the semaphore primitives `P` and `V` are just special cases of the message passing primitives `receive` and `send`.

```
public class Barrier {
  private static final int N = 10; // number of workers
  private static sem proceed;
  private static ProcessGroup WG;
  static {
    WG = new ProcessGroup("Worker Group");
    JR.changeCreationGroup(WG); // all worker processes will belong to process group WG.
  }
  private static process worker( (int i = 0; i < N; i++) ) {
    while (...) { // iterations remain
      // code to implement one iteration of task i
      ...
      // barrier
      P(proceed); // wait for coordinator to say "continue"
    }
  }
  private static op void coordinator() { // no longer a process -- it's invoked on PQ.
    for (int w = 0; w < N; w++) { V(proceed); }
    if (...) // iterations remain
      JR.registerPartialQuiescenceAction(WG, coordinator); // re-register PQ op.
  }
  public static void main(String [] args) {
    JR.registerPartialQuiescenceAction(WG, coordinator); // register PQ op.
  }
}
```

Fig. 6. Barrier synchronization using partial quiescence.

4.2 Key Aspects of Partial Quiescence

As seen in the examples in the previous section, process groups allow the programmer to specify parts of the program for separate PQ detection. The names of process groups, specified by the string argument to the `ProcessGroup` constructor, are in a global namespace. For example, in a multi-VM program (Section 2.2), processes created in process group `"A"` on two different VMs are in the same process group. The programmer can also create a process group specific to a VM by using a per-VM unique identifier in the name.

PQ detection for a process group does not begin until the PQ operation has been registered. This avoids the following "startup problem". Suppose a process group has just been created, but no processes have yet been created within that group, for example, if the main program in Figure 3 registered its PQ operations before instantiating the `MMMultiplier` objects. Then, PQ detection would detect that the group has quiesced, which would not be too useful for the programmer. Just as in GQ, the PQ operation can start up new activity and can re-register another PQ operation.

The precise definition of PQ for JR differs slightly from that given in Section 3. The reason is that in JR a message is sent to an operation, which can be serviced by processes that might belong to different process groups. The PQ definition for JR, therefore, says "(1) there are no messages in the system *that are serviceable by a process in group* A from group A in transit to group A".

A program that uses PQ can be nondeterministic. For example, a message from outside a process group might be sent either before or after PQ is detected for that process group, thus affecting program behavior. However, such nondeterminism does not occur in the examples in this paper (or other practical examples we have written so far). It remains to be seen whether such nondeterminism is a problem in further practice.

PQ is an extension to, not a replacement for, GQ. A program is globally quiescent when all parts of it have become partially quiescent and the remaining processes not associated with any group have terminated or deadlocked, and no messages are in transit. Also, the extended JR has four additional PQ features for more complicated programming situations as illustrated in [12]. First, the program can disable or enable PQ detection features during execution. Second, an optional argument to the process group constructor can specify the number of processes expected in the group; quiescence of the group occurs when that number of processes have terminated or deadlocked. Third, process groups can be hierarchical. A parent group is defined to have become partially quiescent only when all of its child process groups have become quiescent. Fourth, a process can change its process group in the middle of execution.

5 Implementation and Performance

We have an initial implementation of PQ in an extension of JR version 1.00061 (based on Java 1.4). We are presently porting it to JR version 2.00001 (based on Java 1.5).

5.1 Implementation

The implementation of PQ adapts the centralized manager implementation of GQ described in Section 2.2. (The implementation of PQ for closed process groups (Section 3) could follow the GQ implementation rather directly, but with message counts specific to process groups.) When a process group is created on a VM, the RTS (run-time system) on the VM sends a message to the manager. The manager uses the process group name as the key into a hashtable; the hashtable entry contains the list of VMs on which the process group has been created and a capability for the PQ operation. When the PQ operation is registered, it is sent by the RTS to the manager. The manager then creates a thread to handle quiescent messages for this group (if such a thread has not already been created). The thread executes until the group becomes quiescent (as described in the following paragraph), at which point the thread invokes the PQ operation and terminates. If the PQ operation is re-registered, a new thread is created (Exactly when the thread is created is important so that the thread does not detect quiescence before the operation has been (re-)registered, i.e., to avoid the "startup problem" mentioned in Section 4.2.)

When the RTS on a VM detects that a process group on that VM becomes quiescent (i.e., all of its processes have terminated or are waiting to receive a message), it sends an idle message to the manager, where it is handled by the thread that is managing the process group. If the manager has received an idle message for the process group from each VM, it then sends a message to each VM to confirm that the VM is indeed idle. If the manager receives such confirmation, then the process group is quiescent. Otherwise, it waits for idle messages from those VMs who reported they were not idle before it attempts confirmation again. This second, confirmation phase is necessary to account for one VM reporting that the process group is idle just after it sends a message to another process within the same process group on another VM that already reported that it was idle, i.e., to implement the modified PQ definition in Section 4.2.

5.2 Performance

Because PQ is a new language feature, we have no direct basis of comparison to assess the performance of our implementation. However, we have compared the performance of PQ in several programs with the performance of GQ in roughly comparable programs. Specifically, we compared the PQ matrix multiplication program (Section 4.1) with a variant of the original main program in Figure 1 that starts two matrix multiplications (GQ). The results show that over a range of different sized matrices PQ required 0%–1.5% additional time; the multi-VM versions of those programs required 0.4%–4.1% additional time. We also compared the (single VM) PQ and GQ versions of the barrier programs (Figure 6). The times for the two versions over a range of different numbers of workers were always within 3% of each other; the times for a multi-VM barrier program were always within 7% of each other. The PQ version of Figure 6 took 4-10% more time than the (GQ) program in Figure 5. We ran these tests on various PCs (1.4GHz and 2.0GHz uniprocessors; 2.4GHz and 2.8GHz dual-processors) running Linux; specific results, of course, varied according to platform. The actual code and execution times, and further results and explanations appear in [12].

6 Conclusion

This paper introduced the notion of partial quiescence and showed how it can be incorporated into a programming language. Having such a PQ mechanism can lead to a different style of programming, which in some cases is simpler as seen, for example, in the barrier example in Section 4.1. This paper also discussed the implementation of PQ and its performance, which differs only slightly from GQ's performance. Although our early results are promising, further experience is needed with using PQ mechanisms and measuring their costs. In particular, we plan to investigate further concurrent applications and see which might benefit from using PQ mechanisms. We plan to include PQ in the standard JR language release [6].

Acknowledgements

The referees provided thoughtful comments, which helped us to improve our presentation. One referee kindly provided additional references to related work.

References

1. G. R. Andrews and R. A. Olsson. *The SR Programming Language: Concurrency in Practice*. The Benjamin/Cummings Publishing Co., Redwood City, CA, 1993.
2. G. R. Andrews, R. A. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend. An overview of the SR language and implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, January 1988.
3. E. W. Dijkstra and C. S. Scholten. Termination detection for disffusing computations. *Inform. Process. Lett.*, 11(1):1–4, 1980.
4. N. Francez. Distributed termination. *ACM Trans. Programming Languages and Systems*, 2(1):42–55, 1980.
5. J. Helary, C. Jard, N. Plouzeau, and M. Raynal. Detection of stable properties in distributed applications. In *PODC '87: Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 125–136, 1987.
6. JR distribution. `http://www.cs.ucdavis.edu/~olsson/research/jr/`.
7. A. W. Keen, T. Ge, J. T. Maris, and R. A. Olsson. JR: Flexible distributed programming in an extended Java. *ACM Transactions on Programming Languages and Systems*, pages 578–608, May 2004.
8. Y. Kermarrec, L. Pautet, and S. Tardieu. GARLIC: generic Ada reusable library for interpartition communication. In *TRI-Ada '95: Proceedings of the Conference on TRI-Ada '95*, pages 263–269, New York, NY, USA, 1995. ACM Press.
9. N. Kobayashi. A partially deadlock-free typed process calculus. *ACM Trans. Program. Lang. Syst.*, 20(2):436–482, 1998.
10. L. Liang, S. T. Chanson, and G. W. Neufeld. Process groups and group communications: classifications and requirements. *IEEE Computer*, 23(2):56–66, 1990.
11. G. R. Luecke, Y. Zou, J. Coyle, J. Hoekstra, and M. Kraeva. Deadlock detection in MPI programs. *Concurrency and Computation: Practice and Experience*, 14:911–932, 2002.
12. Billy Yan-Kit Man. The design and implementation of partial quiescence in a concurrent programming language. Master's thesis, University of California, Davis, Department of Computer Science, March 2006. `http://www.cs.ucdavis.edu/~olsson/students/`.
13. S. P. Masticola and B. G. Ryder. Static infinite wait anomaly detection in polynomial time. In *Proceedings of 1990 International Conference on Parallel Processing*, pages II.78–II.87, University Park PA, 1990.
14. J. Matocha and T. Camp. A taxonomy of distributed termination detection algorithms. *The Journal of Systems and Software*, 43(3):pp 207–221, 1998.
15. Friedemann Mattern. Global quiescence detection based on credit distribution and recovery. *Inf. Process. Lett.*, 30(4):195–200, 1989.
16. U. Nestmann. What is a 'good' encoding of guarded choice? *Journal of Information and Computation*, 156:287–319, 2000.
17. K. Okano, S. Hattori, A. Yamamoto, T. Higashino, and K. Taniguchi. Specification of real-time systems using a timed automata model with shared variables and verification of partial-deadlock freeness. In *ICPP Workshop*, pages 576–581, 1999.
18. R. A. Olsson and A. W. Keen. *The JR Programming Language: Concurrent Programming in an Extended Java*. Kluwer Academic Publishers, Inc., 2004.
19. Amitabh B. Sinha, L. V. Kalé, and B. Ramkumar. A dynamic and adaptive quiescence detection algorithm. Technical Report 93-11, Department of Computer Science, University of Illinois, Urbana-Champaign, 1993.
20. J. S. Vetter and B. D. de Supinski. Dynamic software testing of MPI applications with Umpire. Technical report, Lawrence Livermore National Laboratory, 2000.