Tool Demonstration: DrHJ — a lightweight pedagogic IDE for Habanero Java

Jarred Payne Vincent Cavé Raghavan Raman Mathias Ricken Robert Cartwright Vivek Sarkar

Department of Computer Science, Rice University

{jrp1, vcave, raghav, mgricken, cork, vsarkar}@rice.edu

1. Introduction

The Java language and runtime environment has had a profound worldwide impact on computer software since its introduction nearly two decades ago. It has enabled the creation of a rich ecosystem of libraries, frameworks, and tools that promises to deliver significant value for many years to come. Consequently, a wide range of Interactive Development Environments (IDEs) have emerged to increase the productivity of Java programmers. They vary in functionality based on the expertise level assumed for their target user base. The Eclipse Java Development Tools (JDT) project offers a rich set of power tools for experienced programmers, but can be harder for novice programmers to set up and use. In contrast, IDEs such as DrJava [2] and BlueJ [16] have been developed primarily for use in introductory programming courses.

In this tool demonstration paper, we summarize the DrHJ tool which will be demonstrated at the conference. In anticipation of the need for introducing parallelism earlier in the Computer Science curriculum, DrHJ extends DrJava with support for the pedagogic Habanero Java (HJ) parallel programming language that was derived from the earlier Java-based definition of the X10 language [4]. DrHJ builds on our past experiences at Rice with developing the DrJava IDE and the HJ language. DrJava is used by many universities world-wide, and has been downloaded over 1.1 million times since its inception in 2002.

The rest of the paper is organized as follows. Sections 2 and 3 summarize the DrJava IDE and the HJ language respectively. Section 4 describes how DrJava was extended to support HJ. In addition to implementing a plug-in extension for HJ in DrJava, DrHJ also includes a data race detection tool for a subset of the HJ language. Finally, Section 5 summarizes current status and future work items for DrHJ.

2. Overview of DrJava

DrJava is a free, open-source lightweight IDE for Java. It is designed primarily for students, providing an intuitive interface and the ability to interactively evaluate Java code in an *Interactions Pane*. It also includes powerful features for more advanced users, enabling (for example) the DrJava team to develop DrJava completely within DrJava.

PPPJ '11, August 24–26, 2011, Kongens Lyngby, Denmark.

The development of DrJava began in 2001, with the first release in Spring 2002 [2]. It was designed to support techniques popularized as "Extreme Programming" [10, 11], *e.g.,* support for test-driven development using JUnit is fully incorporated into the IDE. From the beginning, DrJava supported Java programs that used generic types, which was a novel feature at the time. Later in the evolution of DrJava, support for Java generics was also added to the Interactions Pane.

DrJava's Interactions Pane integrates well with the included source-level debugger and allows users to not only examine and modify variables when a breakpoint is hit, but also to invoke methods and execute complex programs in the Interactions Pane's interpreter. After the addition of a project facility in 2004 and improved support for large projects in 2006, DrJava experienced a sharp increase in popularity.

In 2005, DrJava introduced support for a hierarchy of Java *language levels*, a pedagogic framework that helps beginners learn Java by partitioning the language into levels of increasing syntactic complexity [9]. The language levels are not mere subsets of Java. The levels restrict the use of some Java constructs, such as imperative loops, arrays, exceptions, but they also perform code augmentation by adding necessary modifiers to fields and methods; generating code for constructors; generating code for accessor methods; and generating code for `toString`, `equals`, and `hashCode` methods. Recently, the language level facility has been simplified to provide a more flexible language level that combines the previous elementary and intermediate levels. We call this language level *Functional Java* because it disallows mutation and focuses on computation over immutable algebraic data types.

DrJava is a cross-platform application available for Windows, Mac OS and Linux. The IDE supports several different Java compilers, including Oracle/Sun's JDK, OpenJDK, the Eclipse Java Compiler, as well as research compilers such as NextGen [17], Java Mint [20], and now HJ as well.

DrJava is still under active development by the JavaPLT group at Rice University. Since the inception of the DrJava project, it has been downloaded over 1.1 million times and is being used by many universities world-wide. DrJava has also been used as a teaching tool in books published by Pearson Education and Wiley Higher Education.

3. Habanero Java

The Habanero Java (HJ) language [8] was developed at Rice University during 2007-2010 as a pedagogic extension to the original Java-based definition of the X10 language [4][1]. In addition to its use as a research language in the Rice Habanero Multicore Software research project [7], HJ is used in a new sophomore-level course on

[1] See http://x10-lang.org for the latest version of X10.

"Fundamentals of Parallel Programming" (COMP 322 [1]) which has become a required course for all Computer Science majors at Rice.

The current HJ implementation supports Java v1.4 as its base language, though sequential[2] code in Java 5/6/7 libraries and classes can be called from HJ programs. The HJ runtime system is fully compatible with the latest Java release, but the Polyglot-based [14] HJ front-end does not currently support generics; support for Java generics and annotations in the HJ front-end is currently in progress. The code generated by the HJ compiler consists of Java classfiles that can be executed on any standard JVM.

The HJ extensions to Java are primarily focused on task parallelism. Similar extensions to C and Scala are being pursued in the Habanero C and Habanero Scala projects at Rice. A brief summary of the most commonly-used HJ constructs is included below. (A notable omission due to space limitation is *places* [4, 21], which is used to teach students about data locality and task affinity.) Additional details on HJ can be found in [3] and [1].

1) async: Async is a construct for creating a new asynchronous task. The statement `async` ⟨*stmt*⟩ causes the parent task to create a new child task to execute ⟨*stmt*⟩ (logically) in parallel with the parent task. ⟨*stmt*⟩ is permitted to read/write any data in the heap and to read (but not write) any local variable belonging to the parent task's lexical environment.

HJ also includes support for *async* tasks with return values in the form of *futures*. The statement, "`final future<T> f = async<T> Expr;`" creates a new child task to evaluate `Expr` that is ready to execute immediately. In this case, `f` contains a "future handle" to the newly created task and the operation `f.get()` (also known as a *force* operation) can be performed to obtain the result of the future task. If the future task has not completed as yet, the task performing the `f.get()` operation blocks until the result of `Expr` becomes available. Future tasks are especially well-suited for introducing parallelism in the context of Functional Java.

2) finish: The statement `finish` ⟨*stmt*⟩ causes the parent task to execute ⟨*stmt*⟩ and then wait until all sub-tasks created within ⟨*stmt*⟩ have terminated (including transitively spawned tasks). Operationally, each statement executed in an HJ task has a unique *Immediately Enclosing Finish* (IEF) statement instance [18].

3) isolated: The *isolated* construct `isolated` ⟨*stmt*⟩ enables execution of a statement ⟨*stmt*⟩ in isolation (mutual exclusion) relative to all other instances of isolated statements. As advocated by Larus and Rajwar [12], we use the *isolated* keyword instead of `atomic` to make explicit the fact that the construct supports weak isolation rather than strong atomicity. Commutative operations, such as updates to histogram tables or insertions into a shared data structure, are a natural fit for isolated blocks executed by multiple tasks in deterministic parallel programs. Towards the end of the COMP 322 course, the students are taught how certain patterns of `isolated` statements can be replaced by calls to `java.util.concurrent` (`j.u.c.`) libraries for atomic variables and concurrent collections.

4) phasers: The *phaser* construct [18] integrates collective and point-to-point synchronization by giving each task the option of registering with a phaser in *signal-only*/*wait-only* mode for producer/consumer synchronization or *signal-wait* mode for barrier synchronization. These properties, along with the generality of *dynamic parallelism*, *phase-ordering* and *deadlock-freedom* safety properties, distinguish phasers from synchronization constructs in past work including barriers [6] and X10's clocks [4]. The latest release of `j.u.c` in Java 7 includes Phaser synchronizer objects, which are derived in part [13] from the phaser construct in HJ.

```
finish {
  phaser[] ph = new phaser[n+2];
  for (int j = 0; j<=n+1; j++) ph[j] = new phaser();
  for (int j = 1; j<=n; j++)
  phased (ph[j-1]<WAIT>, ph[j]<SIG>, ph[j+1]<WAIT>) {
    for (iter = 0; iter < NUM_ITERS; iter++) {
      newA[j] = (oldA[j-1] + oldA[j+1]) / 2.0;
      temp = newA; newA = oldA; oldA = temp;
      next;
} } }
```

Figure 1. One-Dimensional Iterative Averaging using Phasers for Point-to-Point Synchronization

(The `j.u.c.` Phaser class only supports a subset of the functionality available in HJ phasers.)

In general, a task may be registered on multiple phasers, and a phaser may have multiple tasks registered on it. Three key phaser operations are:

• *new:* When a task A_i performs a `new phaser()` operation, it results in the creation of a new phaser ph such that A_i is registered with ph in the *signal-wait* mode (by default).

• *registration:* The statement, `async phased` (*ph1*⟨*mode1*⟩, *ph2*⟨*mode2*⟩, ...) ⟨*stmt*⟩, creates a child task that is registered on phaser *ph1* with *mode1*, phaser *ph2* with *mode2*, etc. The child task's registrations must be subset of the parent task's registrations. • *next:* The `next` operation has the effect of advancing each phaser on which the invoking task A_i is registered on to its next phase, thereby synchronizing all tasks registered on the same phaser. In addition, a `next` statement for phasers can optionally include a *single* statement, `next` ⟨*stmt2*⟩. This guarantees that the statement ⟨*stmt2*⟩ is executed exactly once during the phase transition [18, 22].

Figure 1 shows an iterative averaging example to illustrate the power of using phasers for point-to-point synchronization. The for-async pattern creates a parallel loop in which each j-iteration executes as a separate task. Phasers are used to orchestrate interactions among those tasks. In this example, task T_j is registered on three phasers — `ph[j]` in *signal-only* mode and `ph[j-1]` & `ph[j+1]` in *wait-only* mode. Note that phasers gracefully handle boundary conditions that often arise in point-to-point synchronization. For example, the `wait` operations on `ph[0]`/`ph[n+1]` by task $T_{j=1}$ / $T_{j=n}$ becomes a no-op, because there is no task registered on `ph[0]`/`ph[n+1]` with *signal* capability.

5) forall: The statement `forall (point p : R)` ⟨*stmt*⟩ supports parallel iteration over all the points in region `R` by launching each iteration as a separate *async*, and including an implicit *finish* to wait for all of the spawned asyncs to terminate. A *point* is an element of an n-dimensional Cartesian space ($n \geq 1$) with integer-valued coordinates. A *region* is a set of points, and can be used to specify an array allocation or an iteration range as in the case of *async*.

Each dynamic instance of a `forall` statement includes an implicit phaser object (let us call it `ph`) that is set up so that all iterations in the `forall` are registered on `ph` in *signal-wait* mode[3]. Since the scope of `ph` is limited to the implicit finish in the `forall`, the parent task will drop its registration on `ph` after all the `forall` iterations are created.

4. DrHJ

DrHJ is an extension of the DrJava IDE developed at Rice University that supports the HJ language. It was used in laboratory

[2] Some concurrency constructs of Java can interfere with the HJ runtime system; however, we allow the use of non-blocking calls to the `j.u.c.` libraries, *e.g.,* to `ConcurrentHashMap` and atomic variables.

[3] For readers familiar with the `foreach` statement in `X10` and HJ, one way to relate `forall` to `foreach` is to think of `forall` ⟨*stmt*⟩ as syntactic sugar for "`ph=new phaser(); finish foreach phased (ph)` ⟨*stmt*⟩".

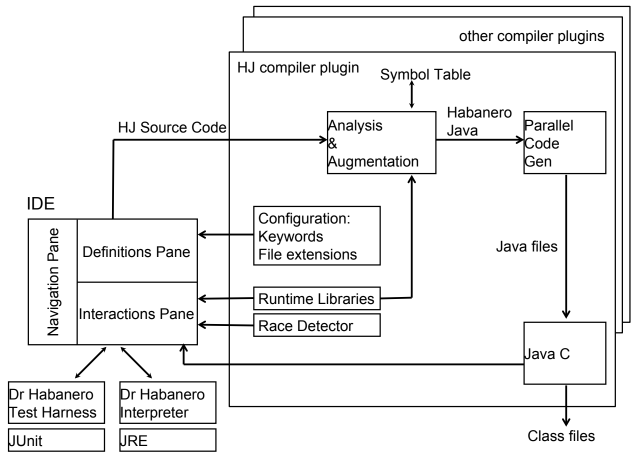

Figure 2. Architecture of the DrHJ IDE

assignments and programming homeworks in the sophomore-level COMP 322 course [1] at Rice. DrHJ is distributed as a single jar file and requires no installation, assuming that a Java Runtime Environment (JRE) is available.

Figure 2 shows the general architecture of DrHJ, which can be divided into two main parts, the graphical user interface (GUI) and a collection of compiler plug-ins. A screenshot of the GUI is shown in Figure 3. DrHJ is composed of three elements: a Navigation Pane that shows documents currently opened (hidden in Figure 3 to conserve space); a Definitions Pane that contains the source code being edited; and a set of bottom panes that includes the Interactions Pane and panes for compiler messages and program output. The Definitions Pane allows users to edit HJ source code and provides syntax highlighting for HJ keywords. Compilation and execution of HJ programs can be done directly in the IDE.

DrHJ provides extension points and a plug-in mechanism that can support multiple compilers. A compiler plug-in is responsible for translating user actions in the DrHJ GUI into invocations of the HJ compiler and runtime. The plug-in configuration specifies various compiler-dependent properties, such as compiler-supported file extensions and a list of keywords for syntax highlighting in the editor.

When an HJ source file is selected for compilation, the HJ compiler plug-in receives a file name from the GUI, as well as information about the source path, the classpath, and the destination directory for the generated class files. The HJ compiler plug-in uses this information to build a list of arguments and then invokes the HJ compiler entry point programmatically. The Polyglot-based front-end [14] of the HJ compiler parses the source file, builds an abstract syntax tree (AST), and performs syntactic and semantic analysis of the code. The AST is then passed to the Soot-based [19] back-end, which transforms HJ constructs into the Jimple intermediate representations, performs code transformations to implement the HJ parallel constructs, and finally generates standard Java bytecode. Compiler error messages are transferred back to the IDE and displayed in one of the bottom panes.

If compilation is successful, the user can invoke the program by pressing the Run toolbar button, or by using the `run` keyword in the Interactions Pane, followed by the name of the program's main class, and an optional list of arguments for the program (for example: `run Fib 10`). As during the compile operation, the IDE passes information about the request to run a program to the currently selected compiler plug-in. The HJ compiler plug-in then builds a list of arguments and invokes the HJ runtime entry point programmatically.

In DrHJ, we distinguish between the main JVM that executes the DrHJ IDE, and the "Interpreter JVM" that executes HJ applications. DrHJ's main JVM communicates with the Interpreter JVM using Java's Remote Method Invocation (RMI) API. When the user instructs DrHJ to run a program, an RMI invocation triggers the execution of the program in the Interpreter JVM. Any output produced by the Interpreter JVM is forwarded back to DrHJ to be displayed in the Interactions Pane. Decoupling program execution from DrHJ provides a better user experience by preventing critical errors such as out of memory conditions from impacting the IDE.

DrHJ also includes a tool to detect data races in HJ programs, based on the ESP-bags algorithm developed for HJ [15]. The ESP-bags algorithm is a generalization of the SP-bags algorithm developed for Cilk's spawn and sync constructs [5]. Like SP-bags, ESP-bags works by following a depth-first execution of a sequentialized version of the parallel program. (The extensions in ESP-bags were necessary because the set of computation graphs generated by async-finish constructs in HJ is more general than the graphs generated by spawn-sync constructs in Cilk.) As a result, the DrHJ data race detector currently supports HJ programs that contain only `finish`, `async` and `isolated` constructs, since those programs can be easily sequentialized. An important property of the DrHJ data race detector is that it uses the depth-first execution to report all *potential* races that may be encountered across all task schedules for a given input.

Figure 3 shows a screenshot of the DrHJ GUI for a simple program *ArraySum.hj*, that attempts to use two tasks to sum the elements of an array. The child `async` task in lines 50–54 computes the sum of elements $X[0 \ldots mid]$ in $X[0]$. The parent task computes the sum of elements $X[mid \ldots n-1]$ in $X[mid]$ in parallel with the child task, and then adds $X[0]$ and $X[mid]$ in line 60 after the `finish` statement which ensures the completion of the child task. However, this code contains an error because $X[mid]$ is read by the child task, and is also read and written by the parent task. This error is detected as a data race by DrHJ, as shown in the Interactions Pane in Figure 3. The error report includes the source coordinates[4] for the two conflicting accesses as well as the index of the array location on which the race occurs.

5. Current Status and Future Work

The DrHJ IDE currently supports the following features as extensions to DrJava:

- Selection of the HJ compiler, which can be either bundled in the same jar file or specified using the `HJ_HOME` environment variable.
- Editing of HJ source files with syntax highlighting for HJ constructs.
- Compilation of HJ source files.
- Execution of HJ programs in the Interactions Pane.
- Race detection option: when enabled, DrHJ compiles and runs HJ programs with data race detection turned on.

Topics for future work include:

- Supporting interpreted HJ constructs in the Interactions Pane. Currently, the Interactions Pane can invoke code (*e.g.,* a method call) containing HJ constructs, but the HJ constructs cannot be interpreted directly in the Interactions Pane.
- Transferring HJ compilation error messages from the console pane to the error pane. DrJava typically displays compiler errors

[4] Support for implementing hyperlinks from the source coordinates in this error message to the source locations in the Definitions Pane is in progress.

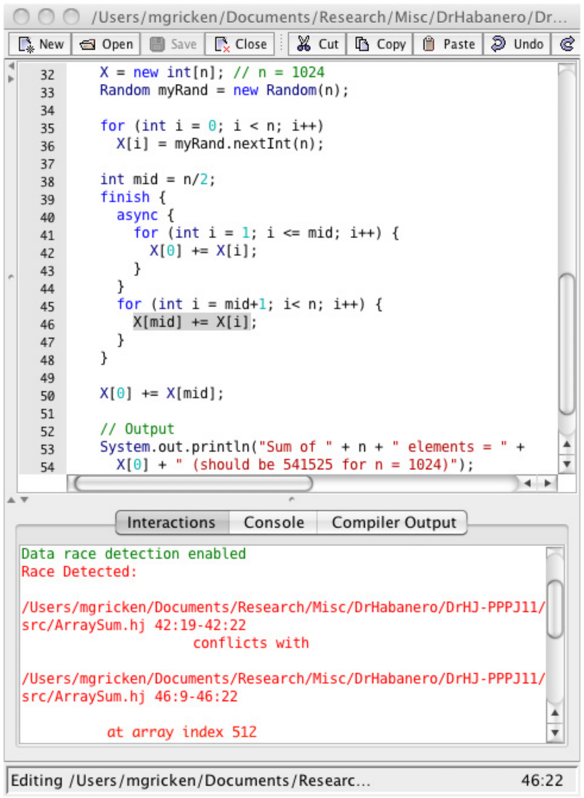

Figure 3. DrHJ GUI screenshot featuring the HJ data race detector

in an error pane, but HJ compilation errors are currently only printed in the console.

- Creating a dedicated Race Detection Pane to display data race errors in HJ programs.

In summary, the combination of the popular DrJava IDE and the high-level HJ parallel programming language enabled us to create a tool suitable for use by undergraduate sophomores in a new introductory parallel programming course at Rice University. The integration of a data-race detection tool with DrHJ provides students with a powerful tool to create, edit, test, and debug parallel programs for laboratory and programming assignments in the course.

References

[1] COMP 322: Fundamentals of Parallel Programming. URL `https://wiki.rice.edu/confluence/display/PARPROG/COMP322`.

[2] Eric Allen, Robert Cartwright, and Brian Stoler. DrJava: a lightweight pedagogic environment for Java. In *Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, SIGCSE '02, pages 137–141, New York, NY, USA, 2002. ACM. ISBN 1-58113-473-8. doi: http://doi.acm.org/10.1145/563340.563395. URL `http://doi.acm.org/10.1145/563340.563395`.

[3] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-Java: the New Adventures of Old X10. In *PPPJ'11: Proceedings of 9th International Conference on the Principles and Practice of Programming in Java*, 2011.

[4] P. Charles et al. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA'05*, pages 519–538, New York, NY, USA, 2005. ISBN 1-59593-031-0. doi: http://doi.acm.org/10.1145/1094811.1094852.

[5] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *SPAA '97: Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 1–11. ACM, 1997. ISBN 0-89791-890-8. doi: http://doi.acm.org/10.1145/258492.258493.

[6] R. Gupta. The fuzzy barrier: a mechanism for high speed synchronization of processors. In *ASPLOS-III*, pages 54–63, New York, USA, 1989. ACM. ISBN 0-89791-300-0. doi: http://doi.acm.org/10.1145/70082.68187.

[7] Habanero. Habanero Multicore Software Research Project web page. http://habanero.rice.edu, January 2008.

[8] Habanero. Habanero Java. http://habanero.rice.edu/hj, Dec 2009.

[9] James I. Hsia, Elspeth Simpson, Daniel Smith, and Robert Cartwright. Taming Java for the classroom. In *Proceedings of the 36th SIGCSE technical symposium on Computer science education*, SIGCSE '05, pages 327–331, New York, NY, USA, 2005. ACM. ISBN 1-58113-997-7. doi: http://doi.acm.org/10.1145/1047344.1047459. URL `http://doi.acm.org/10.1145/1047344.1047459`.

[10] Ron Jefferies. XProgramming.com. `http://www.xprogramming.com`.

[11] Ron Jefferies, A. Anderson, and C. Hendrickson. *Extreme Programming Installed*. Addison-Wesley, Boston, MA, USA, 2001.

[12] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[13] Alex Miller. Set your Java 7 Phasers to stun. http://tech.puredanger.com/2008/07/08/java7-phasers/, 2008.

[14] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An Extensible Compiler Framework for Java. In *Proceedings of the Conference on Compiler Construction (CC'03)*, pages 1380–152, April 2003.

[15] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Efficient Data Race Detection for Async-Finish Parallelism. In *RV'10, Proceedings of the 1th International Conference on Runtime Verification*. Springer, Nov 2010.

[16] Dean Sanders, Phillip Heeler, and Carol Spradling. Introduction to BlueJ: a Java development environment. *J. Comput. Small Coll.*, 16:257–258, March 2001. ISSN 1937-4771. URL `http://portal.acm.org/citation.cfm?id=374685.374841`.

[17] James Sasitorn and Robert Cartwright. Efficient first-class generics on stock Java virtual machines. In *Proceedings of the 2006 ACM symposium on Applied computing*, SAC '06, pages 1621–1628, New York, NY, USA, 2006. ACM. ISBN 1-59593-108-2. doi: http://doi.acm.org/10.1145/1141277.1141656. URL `http://doi.acm.org/10.1145/1141277.1141656`.

[18] J. Shirako et al. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *ICS '08*, pages 277–288, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-158-3. doi: http://doi.acm.org/10.1145/1375527.1375568.

[19] R. Vallée-Rai et al. Soot - a Java Optimization Framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999. URL `www.sable.mcgill.ca/publications`.

[20] Edwin Westbrook, Mathias Ricken, Jun Inoue, Yilong Yao, Tamer Abdelatif, and Walid Taha. Mint: Java multi-stage programming using weak separability. *SIGPLAN Not.*, 45:400–411, June 2010. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1809028.1806642. URL `http://doi.acm.org/10.1145/1809028.1806642`.

[21] Yonghong Yan, Jisheng Zhao, Yi Guo, and Vivek Sarkar. Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement. In *Languages and Compilers for Parallel Computing, 22nd International Workshop, LCPC 2009*, volume 5898 of *Lecture Notes in Computer Science*. Springer, 2009. ISBN 978-3-642-13373-2.

[22] K. Yelick et al. Productivity and performance using partitioned global address space languages. In *PASCO'07*, pages 24–32, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-741-4. doi: http://doi.acm.org/10.1145/1278177.1278183.