DE-FC02-06ER25752: Center for Programming Models for Scalable Parallel Computing
Parallel Programming Patterns

Final Report

Ralph E. Johnson, University of Illinois at Urbana-Champaign
April 2013

Final Report for DE-FC02-06ER25752: Center for Programming Models for Scalable Parallel Computing

Ralph Johnson – University of Illinois at Urbana-Champaign – rjohnson@illinois.edu

This report describes the work done for DE-FC02-06ER25752 at the University of Illinois after I became the PI for the work. At the beginning of the Center, Marc Snir was the PI at Illinois, but he resigned from that position after the first year and I agreed to take over his work with patterns for parallel programming. Prof. Snir has much more experience with parallel programming than I do, but I have long experience with software patterns (with object-oriented programming), and I had been getting increasingly interested in parallel programming patterns.

In addition to working with students and faculty at Illinois, I joined forces with Kurt Keutzer of Berkeley and Tim Mattson of Intel to write a catalog of patterns of parallel programming. We organized a series of workshops on patterns for parallel programming (called ParaPLoP, after the workshops on Pattern Languages of Programming, or PLoP). I helped a group at Microsoft to write a book on how to use some of the patterns for one of Microsoft's parallel libraries.

My goal at the beginning of the project was to develop a catalog of patterns for parallel programming that could capture the heart of parallel programming and that could be used to educate the next generation of parallel programmers. We produced a catalog of parallel programming patterns, and it is useful for teaching parallel programming and for helping describe parallel systems, but it is large and not something that could be put in one volume. We discovered that there are a lot more patterns of parallel programming than there are for object-oriented programming. Parallel programming varies along many dimensions. Object-oriented programming patterns did not depend on algorithms or on machine architecture, but parallel programming does. So, the work to organize parallel programming patterns continues on.

Summary

The purpose of this project was to develop a pattern language for parallel programming. This was mostly to make it easier for people to learn parallel programming, though we also expected that it would improve our understanding of parallel programming and so help to make better tools.

Working with other people, especially Kurt Keutzer at Berkeley, we have done this. However, there were more patterns than we expected, and the resulting pattern language is too large to publish as a single document. So, we are working to publish pieces of it. The first book that has been published so far is aimed at Microsoft platform programmers new to parallel programming.

In addition to the work originally planned, the project funded some work on program transformations related to parallel programming patterns.

The major contributions of this work are:

1) A large pattern language of parallelism
2) A book that introduces some of the patterns to programmers new to parallel programming [1]
3) A particular pattern, *Three Layer Cake,* for how different programming models should be used in a single system [2]
4) Improvement to Photran to make it better support program transformations
5) Cetac, a framework for testing non-deterministic parallel programs [3]

The most important unfinished business is publishing this material in a wider forum.

A brief overview of the work

I started on the project in the fall of 2007. At first I spent a lot of time learning existing work. I studied "Patterns for Parallel Programming" by Mattson, Sanders and Massingill[4]. An important kind of pattern missing from the book seemed to be one based on algorithms (as described by Phil Collela as the "seven dwarves" of HPC) so I started documenting solutions to the N-body problem. I led a seminar in which grad students wrote about the parallel programming patterns that they knew. In the summer of 2008 I met Tim Mattson. He told me that he was working with Kurt Keutzer of Berkeley to make a pattern language for parallel programming and asked me to help. I visited Berkeley twice that fall, and the second time organized a "writer's workshop" to review papers that Berkeley students had written. I invited some members of the patterns community to participate. They had experience with writer's workshops and so helped me teach that style of reviewing.

In the spring, we had regular teleconferences in which we discussed patterns written by the Berkeley students, with a few written by other people. I organized another seminar at Illinois for reading and writing patterns on parallel programming. Some of the students participated in the teleconferences, as well. Gradually these patterns went up on a web site hosted at Berkeley. In May of 2009 we held a patterns workshop, ParaPLoP, in Santa Cruz. About half the people who attended were from Berkeley and Illinois, and about three quarters of the papers were from Berkeley or Illinois.

We continued interacting like this during the 2009/2010 academic year and held another workshop in April of 2010 in Arizona. This time slightly over half the papers were from places other than Berkeley and Illinois. During the 2010/2011 academic year, the group at Berkeley shifted their attention from documenting the patterns to trying to embody the patterns in software frameworks, so they stopped meeting regularly to review papers and our regular teleconferences stopped. We had the third ParaPLoP in April of 2011, though there wasn't any participation by Berkeley.

Current plans are to publish most of the patterns in two volumes. Mattson, Sanders and Massingill will publish a second edition of their book that will be based on the pattern catalog that we have developed. The catalog, of course, started off with patterns from the book. These patterns form the lower levels of the pattern language that we have developed. Mattson, Keutzer and a few of his students are publishing a book that focuses on the patterns at the higher levels. But during 2010 I helped a group at Microsoft write a book that covered some of the patterns from our pattern language[1]. This book was aimed at programmers using Microsoft VisualStudio 2010 to write their first parallel program. Thus, it is only concerned with limited scalability and tends to focus on simple parallelism. Nevertheless, it helps make parallelism accessible to a new set of programmers and does a good job of addressing error recovery, which is too often ignored by parallel programming systems.

My two graduate research assistants have been Nick Chen and Samira Tashirofi. Nick has focused on shared memory parallelism and on low-level optimizations related to the memory hierarchy while Samira has focused on message passing and on testing.

My work on patterns is related to another project of mine on programming environments. Photran is an IDE for Fortran that provides automated refactorings. It is implemented as an Eclipse plugin and is the only open source IDE for Fortran. It supports Fortran 77, Fortran 90, Fortran 2003, and Fortran 2008. It is the basis for a number of my projects on program transformation and on automating patterns. Most of the funding for Photran comes from other sources, but "Programming Models" money was used to fund some undergrads and, in 2011, summer support for Jeff Overbey, who is the main author of Photran. It was also used to fund a postdoc, Munawar Hafiz, who studied the feasibility of using the Photran infrastructure to build a similar system for C and C++ and who worked with Jeff Overbey to build a prototype of a system for C.

Patterns for object-oriented programming are closely related to programming transformation, and I think the same will be true for parallel programming. This theme will be discussed in more detail later in this report.

The next section describes the pattern language of parallelism that we developed with the group at Berkeley. The last section describes work on program transformation.

Our Pattern Language

The patterns are divided into several layers. At the top are patterns that set the context for parallelism. They are patterns about the overall software architecture of the application, or about the kinds of algorithms that the application uses. Our pattern language (abbreviated OPL) calls these "structural patterns" and "computational patterns". The next level down are "parallel algorithm strategy patterns", which are patterns like pipelining, geometric decomposition, data parallelism and recursive splitting. Most of these patterns were in chapter 3 of the book *Patterns for Parallel Programming*. Chapter 4 corresponds to the next level down; Implementation Strategy Patterns, which provide a variety of ways for implementing the algorithm strategy patterns. The bottom of the stack of pattern categories in OPL is Concurrent Execution Patterns, which are supposed to correspond to primitives in the programming languages or platform.

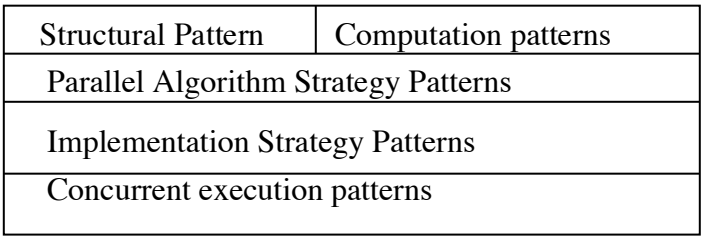


Figure 1: Our Pattern Language

The structural patterns were called "architectural patterns" in *Pattern-Oriented Software Architecture[5]*, and "architectural styles" by Garland and Shaw[6]. They include well-known patterns such as Model/View/Controller, and Pipes & Filters that I have been teaching for years. OPL has added a few that have not been mentioned before, such as Bulk Iteration, which is common in scientific simulation, and refers to systems that load an initial state and then iterate on it until some condition is met.

The computational patterns are derived from the list of "13 Dwarfs" of the Berkeley View[7], which was in turn derived from Phil Collela's list of "7 Dwarfs" of supercomputing. Each pattern is a category of algorithm. Patterns from Phil Collela's list include dense linear algebra and N-body systems, while those added by Berkeley include dynamic programming and graph algorithms.

None of these patterns are about parallelism. Instead, they set the context for parallelism. Some, like pipes and filters, are easy to parallelize. Others, like N-body systems, have a variety of algorithms, each with a different parallelism strategy. Each of them leads to the real patterns of parallelism, the Parallel Algorithm Strategy Patterns, which include patterns like geometric decomposition, data parallelism, and recursive splitting. Because OPL is a pattern language, each of the higher-level patterns should lead to one or more Parallel Algorithm Strategy Patterns, and each of them should have at least one higher level pattern leading to them. So, far, all the higher-level patterns can be implemented in

terms of the Parallel Algorithm Strategy Patterns, which gives us confidence that they are a good set.

OPL does not have patterns about optimization. Parallel programming is used to increase performance, so performance is always important in parallel programs. However, message-passing systems rarely have good performance unless they overlap communication with computation. Shared memory systems rarely have good performance unless they layout memory to increase locality, perhaps by tiling. A complete set of patterns for developing parallel programs has to include information such as this. The Berkeley strategy is to include that information in other patterns, but that can lead to duplication and makes it easy for a pattern author to just omit information about optimization. The Illinois view is that optimizations are a kind of pattern themselves, and should be in the pattern language. Therefore, Illinois people have written several papers on optimizations, such as *Patterns for Overlapping Communication and Computation[8]*, *A Pattern Language for Topology Aware Mapping[9]*, *Patterns for Cache Optimizations on Multiprocessor Machines[10], Ghost Cell Pattern[11],* and *Patterns of Optimized Loops[12].*

One of the questions that is often asked is: how do you decide how to move from a pattern at a higher level to a lower level pattern? My standard answer used to be that each pattern should indicate the patterns that should follow them. This was the way that Christopher Alexander indicated how to move from high-level patterns to low-level patterns[13]. I thought that the reason it was so hard to move from computation patterns to parallel algorithm strategy patterns was that the computation patterns did not have enough detail. Often they were described as a single algorithm, as if they were a kernel, and this is a mistake. They are really classes of problems, and there are many different algorithms for those problems. Our paper *N-body Pattern Language[14]* shows this. It describes several different algorithms for the N-body problem and when you would want to use each one. For each one, it describes the parallel algorithm strategy that works best. The next year *Scalable Sorting Pattern[15]* by Vivek Kale showed the same thing.

But this was not a completely satisfactory answer. High-level patterns are independent of programming model, but the lower level ones are not. Questions like whether to express parallelism using MPI (the SPMI pattern) or OpenMP (The Loop Parallelism pattern) or Fortran array notation (the SIMD pattern) have a big impact on the patterns that are used next, and OPL doesn't describe that well. This led to a pattern that I helped Arch Robison of Intel. This pattern describes how different models of parallelism can fit together.

*Three Layer Cake[2]*

There are many different models of parallel programming. Each has strengths and weaknesses. Programs tend to use several models partly because no one model is good for everything, and partly because the hardware often has features best exploited by one of the models. But these models can conflict with each other. How can we use a variety of parallel programming models in one program and minimize their conflict and

maximize performance, readability, and flexibility?

The solution is to layer these models. Typically there are three layers. The top layer is a message-based model, such as that provided by MPI. It is not good for fine-grained parallelism but can handle non-determinism. In the middle is a shared memory model that accommodates dependencies between data but that works best when there are no dependencies. OpenMP is an example of such a model. On the bottom is a SIMD-like model, such as the vectorization provided by modern C and Fortran compilers. It usually forces determinism. It is both the weakest computational model and the safest. So, I/O and essential non-determinism should move to the top layer, and most computation should move as far down as feasible.

Arch Robison is the author of Intel's Threaded Building Blocks, a C++ library for parallel programming on multicores. So, he is not really interested in MPI and OpenMP, but in tools like TBB that are being increasingly used on desktop machines. Nonetheless, the pattern and the analysis of why this pattern works is as relevant for modern HPC systems as it is for desktop machines.

Two of the patterns that we wrote are part of the concurrent execution patterns. These are *Barrier Synchronization Patterns[16]* and *Collective Communication Patterns[17]*.

In the paper *Reverse Engineering the NAS Parallel Benchmarks: A Parallel Patterns Approach[18]*, Vivek Kale describes the NAS benchmarks in terms of the parallel programming patterns that are in them. This shows that the pattern language is complete enough to describe the benchmarks, provides more examples of the patterns, and provides better documentation of the benchmarks.

The patterns in *Patterns in Testing Parallel Programs with Nondeterministic Behavior*[19] do not really fit into our pattern language, since they are not patterns about parallelizing a program but about what you have to do after you have parallelized it. Nevertheless, they are clearly patterns of parallel programming, so they show the boundaries of our pattern language. In my opinion, our pattern language should include optimization patterns, but patterns like these testing patterns should be considered related, but outside the language.

Even though testing is not part of a delivered application, testing is part of the process of developing it. Testing patterns are therefore about the process of developing an application more than about its structure. But one of the problems with parallel programming is that there are not many good testing tools for parallelism. Samir Tasharofi developed setac[3] as a tool for testing actor programs and discovered the patterns of testing while looking at a variety of test suites for existing systems.

The pattern site at Berkeley is http://parlab.eecs.berkeley.edu/wiki/patterns/patterns

An alternate site at Illinois, which reuses a lot of the material from the Berkeley site but emphasizes the optimizations, is https://wiki.engr.illinois.edu/display/ppp/Home

One of our activities has been collecting examples of parallel patterns. One of the largest set of examples was produced by a group from St. Petersburg University in Russia, but it has turned into an open source project hosted by Microsoft.
http://paralleldwarfs.codeplex.com/

In summary, we've developed a large collection of patterns for parallel programming. It is more than just a catalog of patterns, it has been structured as a pattern language, with high-level patterns that lead to lower-level patterns, helping the designer step through the design process. Our pattern language has been tested on a variety of projects and seems to be fairly complete.

One of the tensions in developing the patterns is how detailed to make them. Should we include patterns for optimizations? How many different algorithms should a computation pattern include? I tend to want to make the language larger and more detailed, while Keutzer tends to want to make it simpler and more abstract. We agree on the basic issue, that making the language larger makes it harder to learn, but that making it simpler means leaving out design information which will be useful for some designers. The problem is that parallel programming has a lot of patterns. The reason it takes a long time for people to learn parallel programming is that there is a lot to learn. Documenting design knowledge as patterns does not change this fact.

From one point of view, our pattern language is already too large, because it is hard for people to learn it all. One approach is to try to hide some of the patterns, perhaps by providing libraries or compilers that let programmers use the patterns without having to completely understand them. My work on program transformation tools could be described from this point of view. A second approach is to find a subset of the patterns that are useful to a particular group of programmers. This is the approach that the Microsoft book has taken. A third approach is to start training programmers in these patterns earlier, to have them learn parts of the pattern language systematically. The second approach is probably the best way to reach professionals, while the third approach might work for students.

Although our pattern language is perhaps too large, it is useful. It is an accurate description of the knowledge needed by parallel programmers. Now we need to focus on packaging it in various ways. This might mean by hiding some of the patterns it inside a library, or packaging a subset of the patterns in a way that appeals to a particular audience, or breaking it down into units that can be taught systematically to students. This is too much for any one group to do, which is why I am eager to work with groups, like Microsoft, who want to use it for their own purposes.

Program transformations

Photran is an IDE for Fortran that provides automated refactorings[20][21][22]. It is implemented as an Eclipse plugin and is part of the standard Eclipse distribution process. Although most of its program transformations are classic refactorings, it also automates some low-level optimizations used to tune a program to a new parallel platform. In other words, it allows a programmer to optimize the program by giving a command to the IDE that causes it to perform a source-to-source transformation, such as unrolling a loop. Photran treats these optimizations like it does the refactorings; it allows programmers to preview them, to undo them, and warns the programmer if the transformation is probably going to introduce an error.

One of our goals for Photran is to investigate the limits of thinking of programming as program transformation, of thinking of all programming as applying patterns. We plan for it to represent a program as a sequence of program transformations. The sequence will not always be an accurate historical record of the creation of the program, but it could be. It will be possible to merge sequences of program transformations and to edit existing sequences. It will be able to represent a program that runs on several parallel platforms as a portable, unoptimized program and a sequence of program transformations for each parallel platform.

Most work on Photran has been supported by other research projects. In the past few years, two grad students have been supported by funds from the NSF and a new NSF project involving Photran started in the summer of 2011. "Programming Models" money was used to support two grad students for a summer, a postdoc for a year, and some hourly undergrads over several years. The undergrads got some experience on a CS research project (several went on to graduate school) and were able to help make Photran more useful for Fortran programmers and a better platform for our research. Some of them worked on providing support for the C preprocessor for Photran (most Fortran programs we have seen use the C preprocessor) and some worked on implementing program transformations[23].

One use of Photran that is closely related to programming models is transforming programs that use MPI to use Adaptive MPI, which is a version of MPI that automates load-balancing. Adaptive MPI (or AMPI) only works on Fortran programs with no global variables, because it runs several versions of a program as different threads within a single process. We used Photran as the basis for a system that changes Fortran programs to eliminate all global state (COMMON, module variables, "own" variables) and to run them with AMPI. A recent paper describes our experience converting FLASH, a large astrophysics simulation. Even though the conversion adds some inefficiencies, and AMPI is a little slower than MPI on programs that are already perfectly load-balanced, the advantages of load balancing made the new version of the system 20% faster than the original. [24]

The object-oriented community considers patterns to be closely related to program transformation, as can be seen by the book *Refactoring to Patterns[25]*. When people

describe patterns, they often tell "before and after" stories; what the system looked like before the pattern was there, and what the system looked like after the pattern existed. In these cases, "applying a pattern" means transforming the system so that it has the pattern. The low-level patterns in our pattern language mostly can be described this way. However, a computational pattern like the N-body Problem does not lend itself to before and after stories. A problem is either an n-body problem or it isn't. However, once you discover that a problem is an n-body problem then there are a variety of solutions you can choose. You can use a naïve $N^2$ algorithm for small problems, a Barnes-Hutt algorithm if the particles are clumped, or a spectral method if the particles are evenly distributed. It is fairly common for a naïve algorithm to be replaced by a more specialized by faster one. This is clearly a program transformation, though not necessarily one easily automated. In general, computation patterns look more like program transformations as they are studied in more detail, which is one of the advantages to making them more detailed.

People using a pattern language try to start with high-level patterns and then derive the system by a sequence of steps, each step applying a lower-level pattern to the system. In practice, the process is not very linear. Designers often have to try several patterns before finding one that works well. Backtracking is common. But when the design is finished, the designer can describe it as a logical progression of steps.

In the same way, a software development project can be seen as a sequence of program transformations. The version control system captures this as a sequence of program versions, each version a transformation of the previous one. Of course, a version control system only captures the transformation as a set of textual differences, it does not understand the reason for the change. It doesn't know whether a particular change is to fix a bug, to apply an optimization, to switch algorithms, or to add a feature. Our vision for Photran is that it *will* understand the reason for the change, at least whether the change is an optimization, a bug-fix, or a new feature. Our first step toward this vision is to build tools that record low-level changes and use them to infer higher-level changes. We used them to measure how often programmers perform refactorings manually even when there are automated versions of these refactorings available[26] and how well version control systems capture low-level detail of editing programs [27]. We are gradually adding support for transformations related to parallel programming and expect that someday tools like Photran will directly support our parallel programming patterns.

HPC researchers interested in program transformation are more interested in C and C++ than they are in Fortran. Much of the work on Photran is language-independent, but it would have more influence if it were more directly applicable to C and C++. During the 2010-2011 academic year, Munawar Hafiz was supported as a postdoc to look at the issues of applying our ideas for program transformation to C and C++. There are many different tools for representing and manipulating C and C++, and he looked at most of them. We decided that it was better to reuse a program analysis system and not to create our own, but none of the program transformation systems were good enough for an interactive refactoring tool. In the end, he worked with Jeff Overbey (the main architect of Photran) to build a C tool using the same infrastructure that was used to build

Photran[28]. Afterwards he went to Auburn University where he runs the OpenRefactory project, a project with Jeff Overbey to make the Photran infrastructure suitable for C.

Parallelism is not the only aspect of software that tends to spread through a program. Others that are hard to modularize include persistence, distribution, reliability, and security[29]. In the long run, a program transformation system that supports one aspect well ought to support other aspects, too. Our work on refactoring started as an attempt to make software more reusable, then it focused more on maintainability, and now we are focusing on performance and parallelism. But all of these are subgoals to bringing our software under control, to eliminate unnecessary complexity and make the software understandable so we can confidently rely on it and easily change it. Patterns and automated program transformations are both means to that end.

Bibliography
An * marks papers and books that I or my students wrote during the course of this project.

[1] * C Campbell, R. Johnson, A. Miller, S. Toub, *Parallel Programming with Microsoft .Net*, Microsoft Press, 2010.

[2] * A Robison and R Johnson, *Three Layer Cake*, ParaPLoP2010

[3] * S. Tasharofi, M. Gligoric, D. Marinov and R. Johnson. Setac: A Framework for Stepwise Deterministic Testing of Actor Programs, The Second Scala Workshop, 2011

[4] T. Mattson, B. Sanders and B. Massingill, *Patterns for Parallel Programming*, Addison-Wesley, 2004

[5] F Buschmann, R Meunier, H Rohnert and P Sommerlad, *Pattern Oriented Software Architcture: A System of Patterns,* Wiley, 1996

[6] D. Garland and M. Shaw, *Software Architectures: Perspective on an Emerging Discipline,* Prentice-Hall, 1996.

[7] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams and K. Yelick, *The Landscape of Parallel Computing Research: A View from Berkeley*, TR UCB/EECS-2006-183, 2006.

[8] * A Becker, R Venkataraman, and L. Kale, *Patterns for Overlapping Communication and Computation,* ParaPLoP'09

[9] * A Bhatele, L Kale, N Chen, and R Johnson, *A Pattern Language for Topology Aware Mapping, ParaPLoP'09*

[10] * N Chen and R Johnson, *Patterns for Cache Optimizations on Multiprocessor Machines, ParaPLoP'10*

[11] * F Kjolstad and M Snir, *Ghost Cell Pattern*, ParaPLoP'10

[12] * S. Tasharofi and R. Johnson, *Patterns of Optimized Loops,* ParaPLoP'10

[13] C Alexander, S. Ishikawa and M Silverstein, *A Pattern Language: Towns, Building, Construction*, Oxford University Press, 1977.

[14] D Dig, R Johnson, and M Snir, *N-body Pattern Language,* ParaPLoP'09

[15] * V. Kale, *Scalable Sorting Pattern*, ParaPLoP'10

[16] * R Karmani, N Chen, BY Su, A Shali, and R Johnson, *Barrier Synchronization Patterns,* ParaPLoP'09

[17] * N Chen, R Karmani, A Shali, BY Su, and R Johnson, *Collective Communication Patterns,* ParaPLoP'09

[18] * V. Kale, *Reverse Engineering the NAS Parallel Benchmarks: A Parallel Patterns Approach*, ParaPLoP'10

[19] * S. Tasharofi, R. Johnson. *Patterns in Testing Concurrent Programs with Non-deterministic Behaviors*, ParaPLOP'11

[20] * J. Overbey, S. Negara, and R. Johnson. *Refactoring and the Evolution of Fortran*. 2nd International Workshop on Software Engineering for Computational Science and Engineering (SECSE'09). Vancouver, Canada, May 23, 2009.

[21] * J. Overbey and R. Johnson. *Regrowing a Language: Refactoring Tools Allow Programming Languages to Evolve*." Onward! 2009. Orlando, Florida, October 25-29, 2009.

[22] * M. Méndez, J. Overbey, A. Garrido, F. Tinetti, and R. Johnson. "A Catalog and Classification of Fortran Refactorings." 11th Argentine Symposium on Software Engineering (ASSE 2010). Buenos Aires, Argentina, September 2-3, 2010.

[23] * J. Overbey, M. Fotzler, A. Kasza, and R. Johnson. "A Collection of Refactoring Specifications for Fortran 95." *ACM Fortran Forum.* 29:3 (2010): 11-25.

[24] * S Negara, G Zheng, KC Pan, N Negara, R Johnson, L Kale, and P Richer, *Automatic MPI to AMPI Program Transformation using Photran*, Workshop on Productivity and Performance (PROPER2010) Aug 2010.

[25] J. Kerievsky. *Refactoring To Patterns*. Addison-Wesley 2004.

[26] * M Vakilian, N Chen, S. Negara, B.A. Raikumar, B. Bailey and R. Johnson. *Use, Disuse and Misuse of Automated Refactorings.* Proceedings of the International Conference on Software Engineering (ICSE 2012), May 2012.

[27] * S. Negara, M. Vakilian, N. Chen, R. Johnson and D. Dig. *Is it Dangerous to Use Version Control History to Study Source Code Evolution?* Proceedings of the European Conference on Object-Oriented Programming (ECOOP 2012).

[28] J. Overbey and R. Johnson. *Generating Rewritable Abstract Syntax Trees: A Foundation for the Rapid Development of Source Code Transformation Tools*. 1st International Conference on Software Language Engineering (SLE 2008). Toulouse, France, September 29-30, 2008. Lecture Notes in Computer Science 5452 (2009): 114-133.

[29] * M Hafiz, P Adamczyk and R Johnson. *Patterns Transform Architectures*.

Proceedings of the 9th Working IEEE/IFIP Conference on Software Architecture, WICSA 2011, Boulder, CO, Jun 2011.