Evolving Equivalent Parallel Programs: Sequences and Iterative Instructions

Conor Ryan and Laur Ivan
{Conor.Ryan|Laur.Ivan}@ul.ie

Abstract

In recent years, the amount of affordable parallel power has increased dramatically,however, current program conversion techniques from sequential to parallel are limited from performance and restrictions points of view. This paper presents an evolutionary approach to program parallelization and is the continuation of the research by [Rya98] [Rya97].

Our system, Paragen, uses genetic algorithms combined with proven transformations for parallelizing the sequential programs. It is designed to tackle both code sequences and iterative instructions. Paragen generates and evolves a set of transformations sequences instead of evolving the original code, by applying these transformations to the original code, the result is a functionally equivalent parallel program.

1 Introduction

The purpose of converting a sequential program to a parallel one is to reduce execution time, while still obtaining the same results as the original program. Two instructions can be executed in parallel if the variables used by one of them are not modified by the other instruction and if neither of them are modifying the same variables.

There are two ways to increase the performance. One choice is to rewrite the entire program for that specific parallel structure. This is usually extremely expensive and involves the employment of highly specialized programmers. The other choice is to pass the program through a parallelization system, which takes as input a sequential program and outputs a parallel version of it.

The parallelization process is a NP-complete problem, therefore, the parallelization tools implement some restrictions for reducing the data dependency. Two trends have emerged within the parallelization tools technique.

One is the combinational approach, which implements algorithms designed to tackle specific program patterns. Its main advantages are that if one of the specific patterns is encountered, the results are very close to the ideal, and the platform specific design offers an optimal solution from an over-head point of view. The disadvantages are that the tool performs poorly with unrecognized patterns as the tools usually are designed for specific platforms; a new module will have to be designed for each new platform.

The other one is the evolutionary approach which tries to evolve a functionally equivalent program from the original sequential one. There are two ways to do this: by evolving the programs and by evolving a set of transformations which are to be applied to the programs.

Our system, **Paragen** belongs to the latter category: it develops a set of transformations which will convert the original sequential program into an equivalent parallel one.

2 Paragen's Environment

Paragen converts the source program into a set of atoms. An atom is defined as a logical program sequence and it takes one *time step* to be executed:

- a simple instruction,
- a set of one or more consecutive loops (*MetaLoop Atoms*),
- a statement block.

The MetaLoop atom was generated in order to define an universal pattern for all categories of loop transformations: the single loop transformations and the multiple loop transformations.

Each atom can be decomposed in atoms at the next nesting level. For example:

```
        Code                 Level 0        Level 1
-----------------------+--------------+--------------
a[0]=10;               |   --->Atom 1 |
for(i=1;i<100;i++) {   |   -+         |
  a[i]=a[i-1];         |    |         |  ->Atom 2.1
  c[i]=c[i]-1;         |    +->Atom 2 |  ->Atom 2.2
  b[i]=b[i]*a[i];      |    |         |  ->Atom 2.3
}                      |   -+         |
m=n+q;                 |   --->Atom 3 |
-----------------------+--------------+--------------
```

Paragen states that each atom takes one time step to be executed. This generalisation is addressed in [She99].

After this parsing, the result is a chain of atoms. At this point, paragen enters the first phase: the Atom Mode. Its purpose is to convert the sequential chain of atoms in an equivalent parallel one. A more detailed description of this mode is presented in the next section.

When processing atoms, paragen encounters MetaLoop atoms. Then it switches into the second phase: Loop Mode. This step's purpose is to optimize the iterative instructions. The loop mode is detailed in the section 4.

Here is an example of how Paragen deals with the above code:

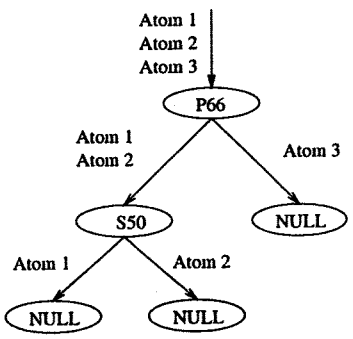


Figure 1: Paragen Tree Example.

- *NULL* represents the tree's leaves and it will generate the data dependency checking and switching to loop mode if necessary.

4 Loop Mode

A loop has cross-iteration data dependencies if, during one iteration, the modified variables need the results provided by other iterations. One of the most common cross-iteration dependencies is:

```
a[i]=a[i-1]+1;
```

The variable `a[i]` needs the result provided by the previous iteration, to which it adds the value "1".

Loop mode consists in applying transformations to the loops. It is designed to follow the same technique as the atom mode: evolve a chromosome of transformations in order to minimize a fitness function. The fitness function measures the gain by transforming the loop, the validity of transformation, the theoretical overhead and is customizable to add other features.

The loop transformations are divided in several categories:

- single loop as input - one loop as output ("PAR")
- single loop as input - multiple loops as output ("Split")
- multiple loops as input - single loop as output ("Fusion" particular cases)
- multiple loops as input - multiple loops as output ("Swap")

This classification requests the MetaLoop atom, which is capable of containing more than one loop for processing.

We have identified over 30 transformations capable of significant loop optimizations, from which most of them are architecture independent. Due to space constraints we reproduce here the most important four.

4.1 PAR

"PAR" just adds the*Parallel execution* attribute for a loop, which means that each of the loop's iteration can be executed in parallel. This is a very powerful transformation. Unfortunately in most of the cases it fails because the loop has cross-iteration data dependencies. Its chances of success increase when used in conjunction with "Split", "Swap", "Fusion".

4.2 Split

"Split" is a technique used for separating the non-parallelizable part from the parallelizable one. It fails whenever there is a post-iteration cross-iteration dependency betveen the instructions within the loop.

```
a[i]=a[i-1]+1;
b[i]=a[i+1];
```

is cannot be splitted because `b[i]` needs the result of `a[i+1]` before it's altered by the next iteration.

4.3 Swap

"Swap" transformation inverts the execution order of two loops by swapping them. Its restrictions are the standdard data dependency restrictions. If there is a data dependency between any of the instructions from the first and second loop, then the transformation fails. This is equivalent to the compression of the loops' instructions into two atoms and comparing them for data dependency.

4.4 Fusion

"Fusion" concatenates two loops into one. Is a technique used for reducing communication overhead (transmitting the data twice over the network usually). In most of the cases, the result is three loops, if the index remapping is unsuccessfull. The best case is the one loop as result case. Generally there is a communication decrease independent of result. This transformation marks negative points when fusioning a parallel loop with a sequential loop because it transforms back the parallel loop into a sequential one.

5 Loop Mode Behavior and Fitness

Loop mode is designed to process a single metaloop as well as multiple metaloops. When processing only one metaloop, all the transformations within the chromosome are assigned to that specific atom. Also, the chromosome has an associated counter which defines the number of transformations applied to the metaloop atom, which makes the transformation vector addressing similar to a cyclic list of transformations (4).

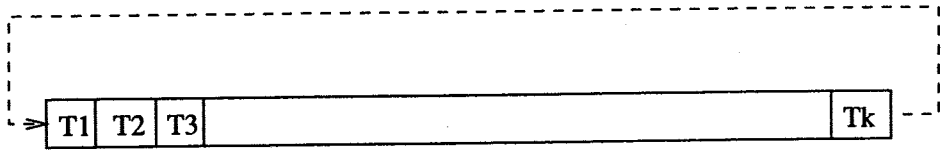


Figure 4: Loop Mode. Cyclic list of transformations. Single Atom

In thins mode, the fitness is the metaloop atom fitness.

In "multiple atoms" mode, paragen assigns a segment from the loop chromosome to each metaloop atom. Each segment reacts as an independent smaller loop chromosome in single metaloop mode, with all its features, including the cyclic addressing (5).A specific metaloop atom is transformed by converting the associated segment into a cyclic list and applying transformations from it. For example, the atom "m" is converted using T_k, T_{k+1}, T_{k+2}, T_k, T_{k+1} ... like transformations.

In multiple atom mode we can differentiate two genetic operators types. The first category acts on chromosome sequences corresponding to the same atom, while the second one acts regardless to the atoms assignment organization of the chromosome. The fitness of the loop atom is the sum of each metaloop fitness.

The *loop mode fitness* is designed to support both boundary defined and undefined loops. If a loop has the boundaries defined, when parallelized, is fairly easy to compute the actual gain:

$$HiBoundary - LoBoundary - 1$$

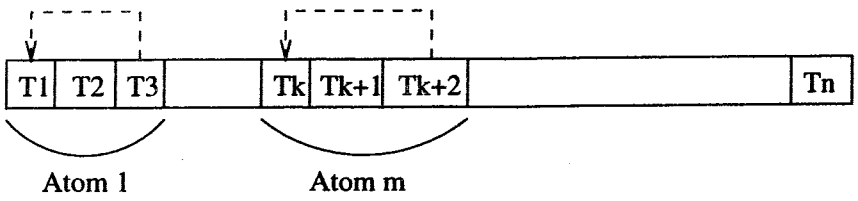


Figure 5: Loop Mode. Multiple atoms. Each atom has a cyclic list of transformations associated

Instead of having *HiBoundary – LoBoundary* loops, the new parallel loop has only one loop spread across *HiBoundary – LoBoundary* processors. This represents the theoretical gain when parallelizing a loop.

If the loop has at least an undefined boundary, then the approach is to count the complexity reductions: if a loop is transformed from `for` to `PARfor`, its complexity is decreased by one.

6 Results

The parallel program's performance has many parameters. The essential criteria are the real speedup and the overhead communication. The overhead communication factor is determined by communicating with the scheduler [She99].

The speedup is often very hard to determine. A common example is when a loop with undetermined iterations is present in the program's code. As mentioned before, paragen states that each atom takes exactly one timestep to be executed. When an atom is such a loop, a synchronization mechanism will wait for it to finish before further executing the code.

6.1 Atom Mode

For illustrating the paragen's performance, we have applied the atom mode to a sequence of 15 data independent instructions. The second problem has the same 15 independent instructions, but repeated twice 6. In each of these experiments, Paragen uses a steady state population of 100, running for 10 generations.

Both cases finished with the discovery of the best possible solution. The second problem is interesting because, although there is a pattern, paragen has no knowledge of this.

6.2 Loop Mode

Loop mode aims to be paragen's main code improvement source. Most programs need to be parallelized because of their iterative sequences and not for the sequential ones. Paragen uses a fusioned method for processing the loop mode: the chromosome integrates information of more loop atoms determined during the atom mode and it is capable of processing all the loop atoms at once. In this mode, the genetic operators take advantage of the chromosome segmentation, which enables the inter-segment mutation and crossover. Here is a test example for a 15 loop atoms sample. All the loops have fixed boundaries. In this situation, the fitness function is measuring the time gain only, instead of mixing with the complexity reduction 7. The original source is listed below in pseudocode.

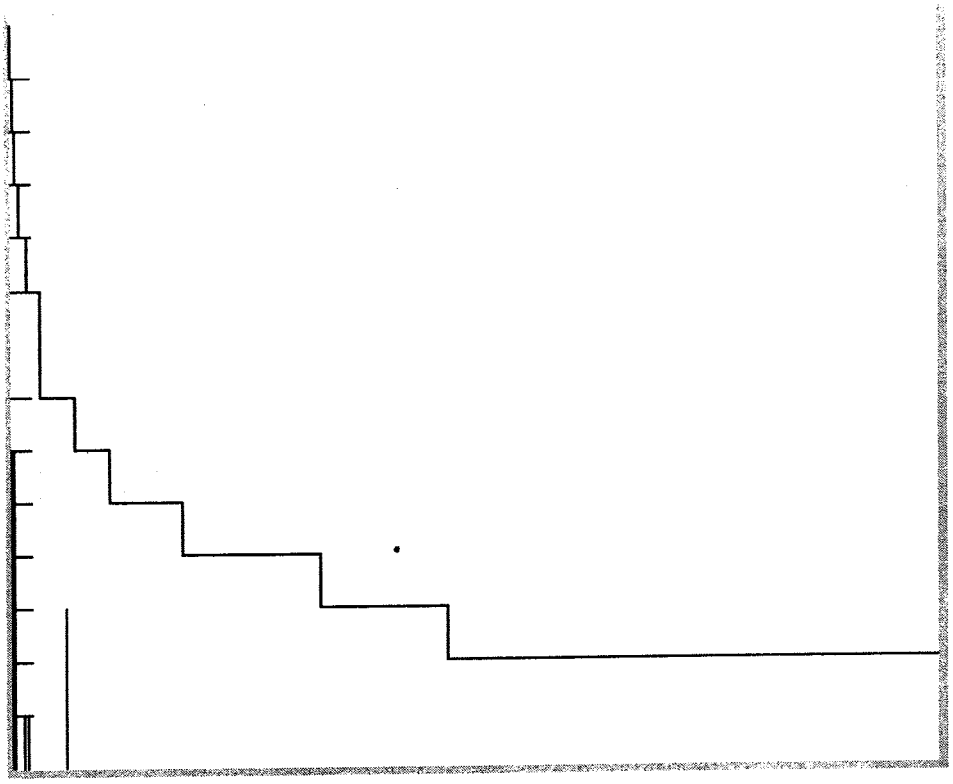


Figure 6: Atom Mode. 15 Parallel Instructions x 2. This is a snapshot of Atom mode GUI, where x axis is number of iterations and y axis is the number of time steps

```
{------------Atom 1------------}
for i=1 to 100
  (c[i+j] ,b) = fun001(b,a[i]);
  (d[i][j],b) = fun002(b,a[i]);
endfor
for i=1 to 100
  (d[i][j],b) = fun003(b,a[i+j]);
  (d[i][j],b) = fun004(b,a[i+j]);
endfor
for i=1 to 100
  (c[i+j] ,b) = fun005(b,a[i]);
  (d[i][j],b) = fun006(b,a[i]);
endfor
for i=1 to 100
  (c[i+j][i][j],b)=fun007(b,a[i]);
  (d[i][j]     ,b)=fun008(b,a[i+j]);
endfor
{------------Atom 2------------}
```

```
for i=1 to 100
  (c[i+j] ,b) = fun009(b,a[i]);
  (d[i][j],b) = fun010(b,a[i]);
endfor
for i=1 to 100
  (d[i][j],b) = fun011(b,a[i]);
  (d[i][j],b) = fun012(b,a[i+j]);
  (c[i+j], b) = fun013(b,a[i]);
  (d[i][j],b) = fun015(b,a[i]);
  (d[i][j],b) = fun016(b,a[i+j]);
endfor
{------------Atom 3------------}
for i=1 to 100
   (d[i][j],b)= fun018(b,a[i]);
endfor
{------------Atom 4------------}
for i=1 to 100
  (c[i+j] ,b) = fun019(b,a[i]);
  (d[i][j],b) = fun020(b,a[i]);
endfor
for i=1 to 100
  (d[i][j],b) = fun021(b,a[i+j]);
endfor
for i=1 to 100
  (d[i][j],b) = fun022(b,a[i+j]);
endfor
{------------Atom 5------------}
for i=1 to 100
  (c[i+j] ,b) = fun023(b,a[i]);
  (d[i][j],b) = fun024(b,a[i]);
for i=1 to 100
  ([d[i][j],b)= fun025(b,a[i+j]);
{-------Atom 6 = Atom 3--------}
{-------Atom 7 = Atom 4--------}
{-------Atom 8 = Atom 5--------}
{-------Atom 9 = Atom 3--------}
{-------Atom10 = Atom 4--------}
{-------Atom11 = Atom 5--------}
{-------Atom12 = Atom 3--------}
{-------Atom13 = Atom 4--------}
{-------Atom14 = Atom 5--------}
{-------Atom15 = Atom 3--------}
```

The testing results also show a dramatic increase of performance with the population increase, while the speed decrease is slow.

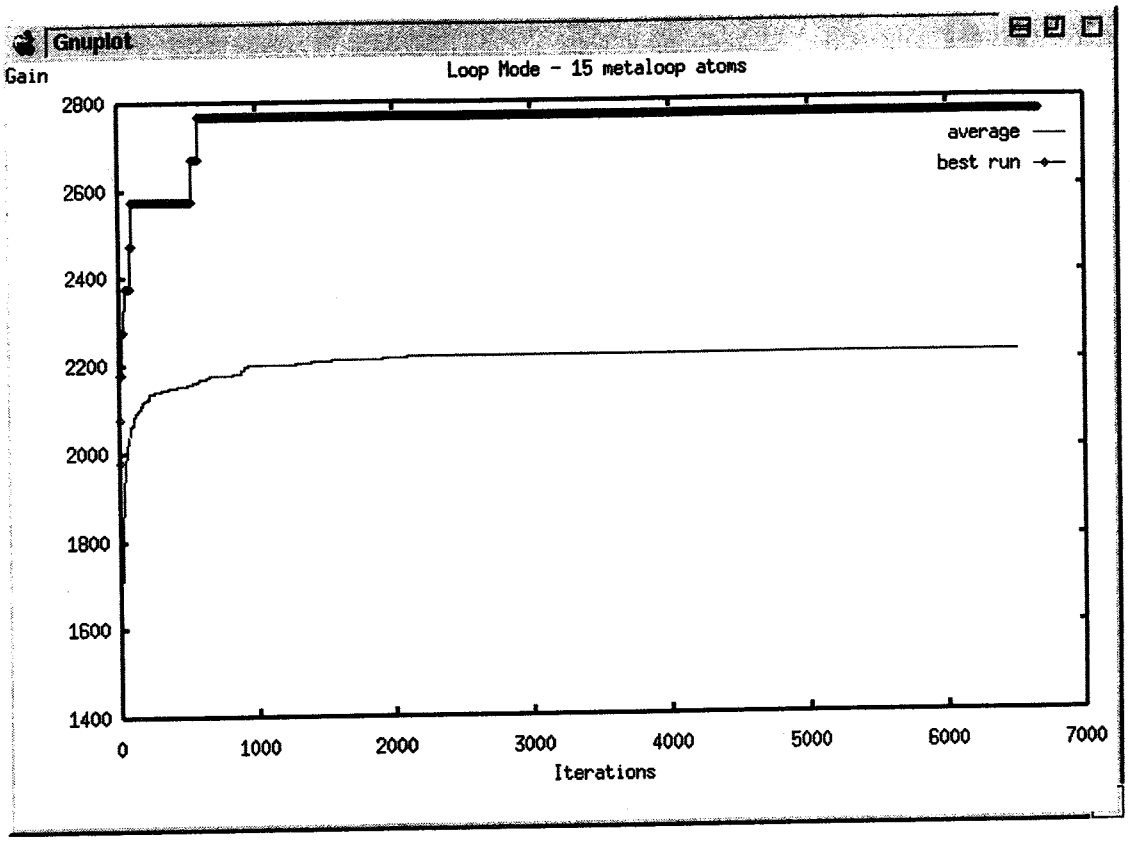


Figure 7: Loop Mode. 15 MetaLoop atoms

7 Conclusion and Future Work

This paper has presented preliminary results of atom mode as well as some tests based determinations for the loop mode. Due to space constraints, we have presented only a sample of the discovered loop transformations.

The loops optimization module is the key to a better performance for any compiler. This is why a great deal of the undergoing work is to locating more loop transformations and to integrating them in the loop mode's structure, with the designing and implementation of the switching engine.

Another part of the research includes the implementing different breeding strategies and testing Paragen against other parallelization systems, as well as testing Paragen along with various parallel compilers.

References

[She99] Sheahan Alan, Collins J.J., Ryan Conor (1999) A Genetic Clustering Algorithm. SCASE Workshop Proceedings, 1999

[Bur88] Burns, A. (1998): Transforming Occam Programs. In Programming Occam 2 . Addison-Wesley

[Dav91] Davis, L.V. (1991): Handbook of Genetic Algorithms. Van Nostrand Reinhold

[Lew92] Lewis, T. (1992) : Introduction to Parallel Computing. Prentice Hall

[Rya98] Ryan, C. and Ivan, L. (1998): Automatic Parallelization of Loops in Sequential Programs using Genetic Programming. GP 98

[Rya97] Ryan, C. and Walsh, P. (1997): The Evolution of Provable Parallel Programs. GP 97

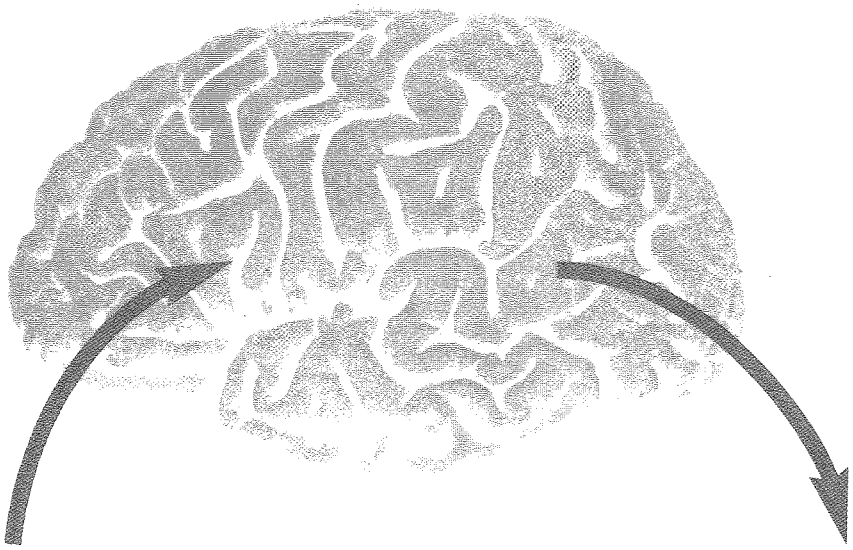
[Rya96] Ryan, C. and Walsh, P. (1996): Paragen: A novel technique of the Autoparallelization of Sequential Programs using GP. In Genetic Programming. MIT Press

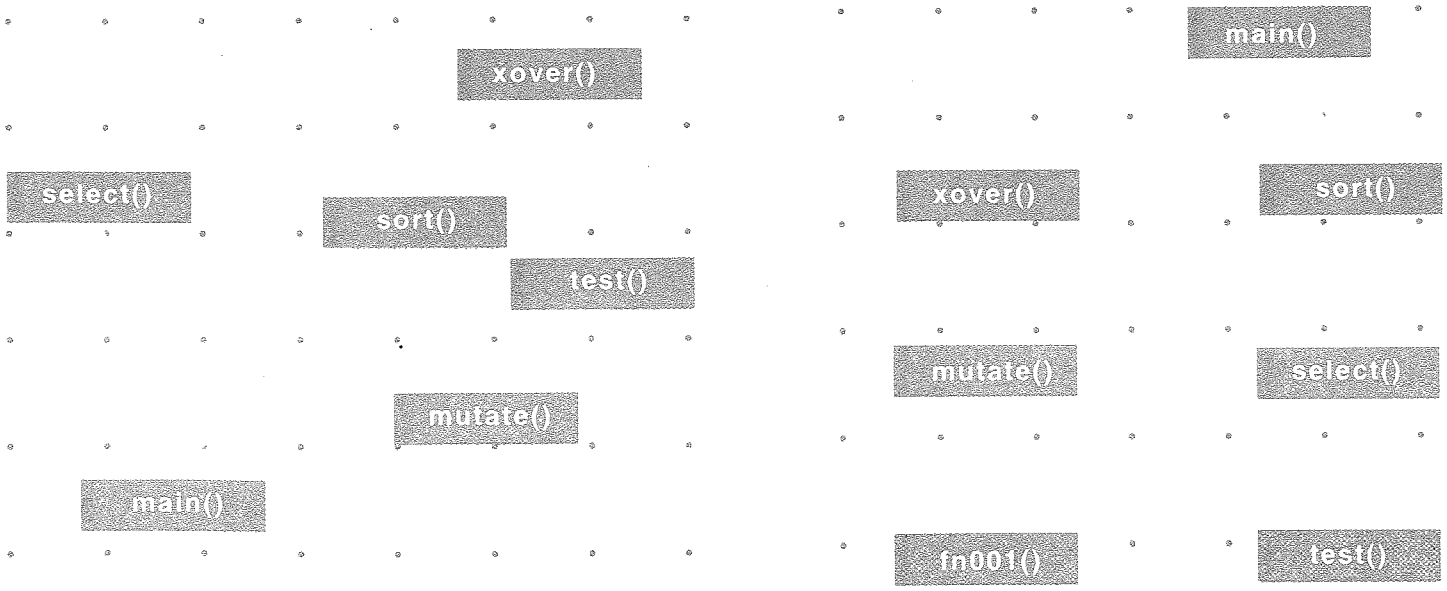
[Rya95] Ryan, C. and Walsh, P. (1995): Automatic conversion of programs from serial to parallelusing genetic programming. In proceedings of Parallel Computing. Springer-Verlag

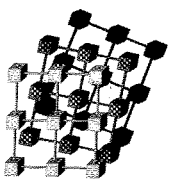
SCASE'99

SOFT COMPUTING APPLIED TO SOFTWARE ENGINEERING

UNIVERSITY OF LIMERICK, IRELAND
APRIL 1999







edited by Conor Ryan and Jim Buckl

SCASE'99

Proceedings of the 1st International Workshop on Soft Computing Applied to Software Engineering

Limerick, Ireland.
12th-14th April, 1999

Editors:

C. Ryan and J. Buckley

ISBN: 1-874653-52-6
Limerick University Press.

SCARE
Soft Computing and Re-Engineering Group.
University of Limerick, Ireland.
Funding by Forbairt.