An Object-Oriented Approach to Software Restructuring

PAUL SAGE AND PETER MILLIGAN
Department of Computer Science
The Queen's University of Belfast
Belfast BT7 1NN

Abstract: - The past ten years has seen a rapid increase in the availability of multiprocessor architectures, both tightly-coupled and network oriented. However, the full or even partial potential of these new platforms has not been realized due to the lack of support for code development. This has encouraged research aimed at the provision of a number of integrated development environments or IDE's which assist in either code development from scratch or the migration of existing or legacy system codes. This paper overviews the problem of source to source translation with respect to both sequential to sequential and semi-automatic parallelisation by introducing an object model which aims to encapsulate the essence of code restructuring.

Key-Words: Object-Oriented Restructuring Parallelisation IMACS/IEEE CSCC'99 Proceedings, Pages:3291-3296

1 Introduction

During the past several years there has been a rapid growth in the development and availability of parallel architectures. Not all of this growth has been restricted to the development of tightly coupled platforms provided by the major hardware manufacturers. Enhancements in networking technologies have supplemented the more traditional multiprocessor architectures with the emergence of distributed workstation clusters. However, it is clear, that the availability of development environments for distributed software has not kept pace with hardware releases, largely due to the fact that parallel software is considerably more difficult to develop than more conventional sequential code. In addition, many of the users for such architectures are firmly based within the science and engineering community, and have traditionally relied on existing or "dusty deck" codes to support their research. The implicit difficulty in re-implementing these codes to take advantage of new hardware has engendered a reluctance to "re-invent the wheel".

Consequently, there is a clear need for the provision of "enabling" technology in the form of a suitable development environment for parallel software, as flexible and as easy to use as those which are readily available for the construction of sequential codes, such as the Mircosoft Development Studio suite of tools.. Ideally, such an environment should enable the programmer to reason about a computational problem in an abstract manner without immediate (and sometimes limiting) reference to architectural issues.

In 1991 the FortPort project[1][2] was initiated at Queen's to examine the automated migration of Fortran codes (Fortran77) to multiprocessor architectures. The system was based on a partial compiler model which, through the processes of lexical and syntax analysis, could generate a syntax graph representation [6] of a source program. When supplemented with data gained from data dependence, "hot spot" analysis and loop characterization, program loops could be transformed to remove the impediments to parallelisation and subsequently partitioned. This process was guided through use of an expert system containing data and rules that, on the basis on loop characterization data, would suggest a transformation pattern for a given loop.

While this approach proved successful the implementation suffered from a number of limitations, the most fundamental of which largely restricted the system to the analysis of Fortran77 codes. The syntax analyser of FortPort was based around a recursive descent analyser with a hard coded representation of the production rules [3,4] defining the source language. Any desire to accommodate a different source dialect would, therefore, require a re-implementation of the syntax analyser. The static nature of the syntax analysis process has a knock-on effect on subsequent processes such as data dependence analysis and program transformation in that they are based around the manipulation of inherently Fortran structures. These

limitations have been addressed in the wider context through the development of "parser generator" tools such as Lex and Yacc aimed at generalizing the source program decomposition process. These tools, when provided with a definition of the source language in the form of production rules, can produce a breakdown of a given source program, which does, however, remain closely associated with the source language.

While previously, the system only considered the specific problems associated with the restructuring of sequential codes, the knowledge and experiences gained in the project phases have been used to expand the analysis domain to examine the general problem of source to source program restructuring. For example, the issues underlying the translation of say, Cobol programs to Java programs, can now be reasoned about in an effective manner.

This paper introduces an object-oriented model which aims to address these problems by separating process from data to provide a more generic framework for code restructuring.

2 A Modified Approach

To overcome the limitations imposed by the static nature of FortPort and introduce a more flexible approach to code restructuring, an object-oriented design methodology. OMT [10][12], was applied to the existing processes. This has yielded an updated process model which moves away from a language-dependent program representation, in the form of a syntax graph to a more abstract, object-based program model (Language Independent Program or LIP). This model aims to encapsulate the statement dependencies and semantics of a program without direct reference to the programming language in which it was originally encoded. This approach enables the expansion of the system to consider a range of source languages, thereby establishing a framework for general code reengineering.

Three main processes are defined in the general code restructuring model:

Parse

This process defines the translation of a source program in a given language to the LIP form:

Parse : source-language-definition X source-program -> LIP instance

Transform

This process incorporates methods for application-specific code reengineering such as restructuring for execution on a parallel architecture, or for sequential to sequential code translation between distinct languages such as Cobol and Java:

Transform : LIP instance -> Lip instance

Generate

This process involves the generation of language specific (and sometimes architecture specific) programs from the language independent form:

Generate : LIP instance X source-language-definition -> source-program

The Language Independent Program, together with each of the programming languages which are used in conjunction with the system, is defined by the following class hierarchy:

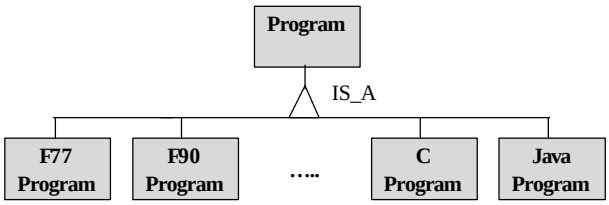

Figure 1 : Language Hierarchy

The program class represents the constructs common to all programming languages and, when instantiated, represents a single program in a language independent form. The definition contains a knowledge base, which contains both data and rules governing the syntax of a "generic" program. The knowledge base is extended through sub-classing and contains "purpose specific" information relating to the restructuring process. For example, the generic knowledge base contains, among other, the rules and data which guide the following processes:

(i) the application of transformations for data dependence reduction, and
(ii) data and functional partitioning for the representation of a parallel algorithm.

These are defined in terms of general program structure. Specific programming languages are represented as subclasses of class program, from which they inherit behaviour. Each programming language class contains a knowledge base defining the production rules of the specific language, which is used in both the parsing and code generation processes. This data is represented as static or class specific data and is consistent and available for used by any given programming language instance.

A Language Independent program may be created from, say, Fortran77 or Cobol using the following code fragments:

```
Program p = new F77Program(sourcefile.f77,
errorfile) ;  // or

Program p' = new CobolProgram(sourcefile.cbl,
errorfile) ;
```

Variable ***p*** is a reference to an instance of class Program, but more specifically, in accordance with the rules of object-orientation, refers to an instance of a class which is derived from Program, which in this case is F77Program. In this way ***p*** can be used to refer to a program in specific terms (i.e. the Fortran77 dialect) or in general terms. The parsing of the source program is implicit in the construction of the F77Program object and produces both the language specific and general representations of the program. Program analysis techniques such as Hot Spot analysis [7] and data dependence analysis [8] may also be performed as actions of the construction event.

The proportion of the original source program represented in general terms is determined by how much of the original syntax, as represented by the programming language production rules, is written in terms of the LIP syntax. Redefinition of a given language syntax is, indeed, a non-trivial process. At the moment only a portion of the constructs of a language such as Fortran77 is represented in LIP terms. To facilitate existing research the redefinition is largely restricted to loop structures with simple statement sequences. This does, however, enable the study of data dependence reduction and partitioning to proceed within the new framework. Indeed, this approach has increased the scope for the analysis of feedback on knowledge assisted transformation. In addition to testing the outcome of a transformation process by executing the generated code on actual hardware, it has become possible to consider the simulated execution of a Language Independent Program on an appropriately model abstract architecture. The inclusion of the correct software metrics within such an abstract definition could produce a "rule of thumb" guide on the potential speed of a fragment of code with a given transformation history, in a fraction of the time that it would take to execute in reality. The structure of the language independent program or LIP is discussed in the following section.

3 The Language Independent Program

The language independent program, LIP, is an object designed to encapsulate the nature of programs in a generalized but simple fashion. The LIP object describes a program (See figure 2) as a single entity which is composed of a sequence of statements with a well-defined thread of execution.

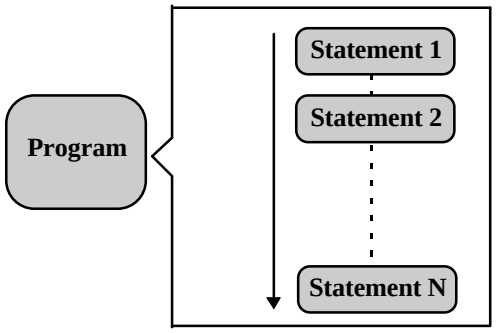

Figure 2 : Program Representation

A program statement is represented by a Statement object that can have a number of subtypes defining the types of statement found in any programming language. Under current definitions (See figure 3) a Statement object may be one of the following:

- An Iteration Statement or Loop Statement,
- A Selection Statement,
- A Statement block,
- An Assignment Statement

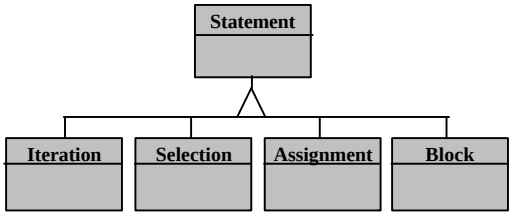

Figure 3 : The Program Statement Hierarchy

An instance of the Iteration Statement class defines a program loop with instance variables describing the iteration space (either in absolute terms or as a linear function of integer variables used to control the loop) together with the loop body, which is represented as a sequence of Statement objects. The Selection Statement class is used to instantiate objects representing alternative program paths, with choice based on evaluation of an associated boolean expression. A Statement block object is simply another statement sequence treated as a single entity. Assignment Statement objects represent simple assignments of the form:

```
Variable = expression ;
```

where Variable represents a simple program variable of a primitive type or an array element of primitive type, and expression is limited to a simple expression formed by combining primitive variable with a range of logical and mathematical operators.

Data dependencies, which inhibit concurrency, are maintained within the program structure at the statement level through a "dependency association" object, which describes the nature of a dependency between two statements.

4 Expressing Explicit Concurrency

In addition to dependency information defining potential parallelism the LIP can be extended to represent a parallel program model. The extended model describes a parallel program in general terms without reference to any particular architecture and aims to be as general and as useful as the sequential programming model for von Neumann architectures.

The von Neumann model assumes that a processor can execute sequences of instructions which not only specify the thread of control, but the addresses of data to be read from and written to. It is possible to program in terms of this sequential model by writing appropriate machine language but this approach has long since been replaced by modular design techniques: programs are structured from simple components which are, in turn, expressed in terms of higher-level abstractions such as loop statements, selection statements and procedure calls. These abstractions help the exploitation of modularity as objects can be manipulated without concern for their internal structure. Sequential programs expressed in this form may be easily translated into executable code. Production of programs for parallel architectures introduces more complexity and, therefore, the concepts of abstraction and modularity are at least as important as in sequential programming.

To be of practical use the model must satisfy four fundamental requirements for parallel software, namely:

(i) modularity
(ii) scalability
(iii) locality
(iv) concurrency

A suitable model for the explicit definition of concurrency and locality is the task / channel model defined in [9]. In this model a parallel computation consists of one or more tasks which can execute concurrently. Tasks are connected by channels which represent inter-task dependencies. A task instance encapsulates a single tread of execution (a program in the LIP definition), together with some local stack memory. The task abstraction provides a mechanism for reasoning about locality: data within a task's definition is considered "close" as opposed to "external", remote data. Tasks can be mapped to physical processors in various ways; the mapping technique employed does not affect the semantics of the program.

5 Advantages of the OO Approach

There are three main reasons for the application of object-oriented modeling techniques in the life cycle of a software project:

(i) ***encapsulation*** : enables the identification and containment of system objects and defines their relationships;
(ii) **separation of process from data** : an object's interface defines its usage without reference to its data representation;
(iii) ***code reuse*** : general behaviour can be packaged and reused to define addition classes.

Processes which operate on the basis of object comparison can be defined in general terms, without any knowledge of how data is represented within a given object. For example, the code transformation process of FortPort [8,9] is driven by a knowledge-based system with rules defining the conditions under which program transformations may be selected and applied. A given rule may be guarded by, what amounts

to, a boolean expression involving some form of data comparison. In the case of loop restructuring, the comparison is between the characteristics of an actual loop and characteristics governing the application of a transformation.

The encapsulation mechanism of object-orientation enables the definition of interfaces which describes the general behaviour of an object. A class defining loop characteristics is essentially defined in the following way:

```
class LoopCharacteristics
{  // public & private instance & class
variables

   boolean compare1(LoopCharacteristics lc)
   {  // method implementation
   }
      :

   boolean compareN(LoopCharacteristics lc)
   {  // method implementation
   }
}
```

Instances of class LoopCharacteristics are then used in the code transformation process without reference to specific loop characteristic data, or method implementation:

```
// assuming A and B are instances of class
// LoopCharacteristics
   :
   if ( A.comparisonMethodX(B) )
      fire associated rule ;
```

where 1<= X <= N

5 Conclusions

Abstraction tools such as object orientation provide, not only a mechanism for modeling the objects which compose the structure of a system, but facilitate a clear description of process. This has enabled the creation of a flexible framework for the analysis and restructuring of both sequential and parallel codes. Research to date has been accommodated within this framework which has encouraged the reapplication of process specific techniques for parallelisation to the general area of source program to source program restructuring. The initial studies on the representation of a given program in LIP form has also encouraged some thoughts towards the definition of an associated programming language syntax called LIPS and the development of an abstract machine on which LIPS programs may “execute”. At the moment research in this area is concerned with simulating the execution of LIPS programs on the abstract architecture with the view to obtaining profile data governing “rule of thumb” performance. It is anticipated that this data will be used to drive the restructuring process for say, automatic parallelisation.

References:

[1] R.K. McConnell, P. Milligan, S.A. Rea and P.P. Sage, FortPort: A Migration Tool for Fortran Codes IMA Conference on Parallel Computation, Oxford, September 1991

[2] P.Milligan, R.K. McConnell, S.A. Rea and P.P.Sage FortPort: An Environment for the Development of Parallel Fortran Programs Microprocessing and Microprogramming, Vol 34, 1992, pp 73-76

[3] S.A. Rea, P. Milligan, P.P. Sage and R.K. McConnell, Porting ‘Dusty Deck’ Fortran Codes to a Multiprocessor Environment Proceedings of the PLUG III, April 1992, pp 46-55

[4] P.Milligan, R.K. McConnell, S.A. Rea, T.J.G. Benson and P.P. Sage, Apparently Sequential Programming Environments for Parallel Computing Parallel Computing and Transputer Applications, ISO Press/CIMNE, Barcelona, 1992, pp 297-306

[5] P.Milligan, T.J.G Benson, R. McConnell, A. Rea and P.P. Sage, Detecting Components for Parallel Execution within the Mathematician’s Devil Microprocessing and Microprogramming, Vol 37, 1993, pp 65-68

[6] P.P. Sage, p. Milligan, R. McConnell, A. Rea and M.T. McCarney, Graph Management Within the FortPort Migration Environment Microprocessing and Microprogramming, Vol 37, 1993, pp 137-140

[7] R.McConnell, P.P. Sage, P. Milligan, A. Rea and P.J.P. McMullan, Hot Spot Analysis within the FortPort Migration Tool for Parallel Platforms Microprocessing and Microprogramming, Vol 37, 1993, pp 141-144

[8] P.Milligan, P.P. Sage, P.J.P. McMullan and P.H. Corr, A Knowledge Based Approach to Parallel Software Engineering

Software Engineering for Parallel and Distributed Systems, Chapman and Hall, April 1996, pp 297-302, ISBN 0-412-75640-0

[9] P.J.P. McMullan, P. Milligan, P.P. Sage and P.H. Corr, A Knowledge Based Approach to the Parallelisation, Generation and Evaluation of Code for Execution on Parallel Architectures IEEE Computer Society Press, 1997, pp 58-63, ISBN 0-8186-7703-1

[10] Smith FJ, Tripathy SR, Sage P. An Object Oriented Approach to Material Selection, 13th International CODATA Conference on New Data Challenges in our Information Age, Beijing, China, Ed. Glaeser & Millward, 1992, pp. A72-A83.

[11] Smith FJ, Sage P. The intelligent Selection of Materials from a Design Specification, Abstract 14th International CODATA Conference, Chambery, France, 1994, pp. 132.

[12] Smith FJ, Krishnamurthy MV, Tripathy SR and Sage P., An Intelligent Object Oriented Database System for Materials Information, Computerization and Networking of Materials Databases, Vol. 4, 1995, pp. 183-193.