Chapter - 11

Conclusion and Future Work

In this thesis we have investigated a suitable process model and its refinements for distributed and object-oriented programs to serve as a suitable intermediate representation for efficient slicing. Such a slicing scheme can have a large number of applications e.g. program debugging, understanding, parallelization, etc. We have taken up only one application (i.e. debugging) for investigating the effectiveness of our approach.

After defining the term hierarchical slicing, we have extended the Program Dependency Graph for sequential programs to include Object Orientation and Multi-threading that supports hierarchical slicing. The extension is natural in that we can use the existing two pass slicing algorithm on the resulting dependency graph. Multi-threading issues are then analyzed. We have used this representation for the construction of a static version of a slicer for a subset of the Java grammar having all the necessary features such as inheritance and multi-threading.

The OPDG provides a static representation to object-oriented programs. Object-oriented systems have a dynamic nature that is more complex and different that the static one. The OPDG is created at compile time,
representing important run-time issue like dynamic binding with special notations. We discussed the construction of an intermediate representation of object-oriented program such as C++ based on three layers (CHS, CDS, DDS). We also proposed a slicing algorithm for computing slice on the proposed OPDG and illustrated its working with an example.

We have prosed a hierarchical intermediate representation for concurrent programs. The different levels of this hierarchy are the process graph, concurrency graph and OPDG. The intermediate representation can handle both message passing as well as shared memory-based interprocess communications. The hierarchical intermediate representation enables computations and display of slices at different levels of abstraction. This feature has been found by us to be very useful in debugging and program understanding. As an applications of our slicing techniques, we have examined the problem of reverse execution of programs written in high-level languages like C, C++, and Java. Given any program state during execution of a program, we have examined an efficient approach to support "undo" operation which restores the machine to a previous state of execution. The reverse execution implies undoing each of the instructions in the reverse sequence. The problem has been considered by defining inverses of statements and maintaining a trace file. We have seen that meaningful inverses can be defined only for a very small subset of program statements. The operations which perform one-to-one or one-to-many mappings have inverses. Therefore, for other statements we must
maintain the value transformations achieved by those statements in a trace file in order to do a reverse transformation during backward execution.

We have designed and implemented a bi-directional debugger for handling distributed C programs and Object Oriented Programs in Java in either direction. For statements which do not have inverses, the value can be maintained by using a check pointing technique. The goal of our design of the bi-directional debugger is to minimize the amount of information to be recorded in a trace file.

The bi-directional debugger does not recover if illegal accesses are made by the variables. But the bi-directional debugger can diagnose logical faults in the program. The facility of single-stepping in either direction enables the user to traverse through the program and examine values of the variables of interest and detect the exact point where the program is misbehaving.

Though similar in many ways the bi-directional debugger for distributed and object-oriented programs differ in many aspects. The debugging of the sequential elements of both kinds of programs are handled in the same way except for handling of objects which forms a part of the bi-directional debugger for the object-oriented programs. The parser for object-oriented program has some extra facilities in addition to producing inverse of the statements. It also extracts all information about the class and return them to
the executive. The difference lies in the implementation of the interpreters for both the case. This is so because we are dealing with two programming languages having diverse features. Nevertheless, as the bi-directional execution is based on the same philosophies, the performance is comparable.

11.1 Scope of Future Work

A design of such debugger as one described in the thesis is highly desirable. Intense research is needed to handle parallel programs, using various other models of interprocess communications such as shared memory. Any work towards achieving bi-directional debugging at the system level (debugging of system executable files) will have immense impact on programming and will prove more than an indispensable tools to different class of programmers.

Following points can be taken up future work:

- Dynamic slicing of an current object-oriented programs can be investigated which uses the system run-time information along with the compile time information.
- Accuracy can further be enhanced using tokens as the vertices instead of statements (consisting of many tokens) to be a node in the PDG.
- Another possible future work is to write a complete pseudo dynamic slicer for Java, which can handle all types of concepts of a multi-threaded language. We have not implemented many features of slicing which we did not consider crucial to the demonstration of the basic
concepts of the bi-directional debugger. We do not, for example, implement interfaces.