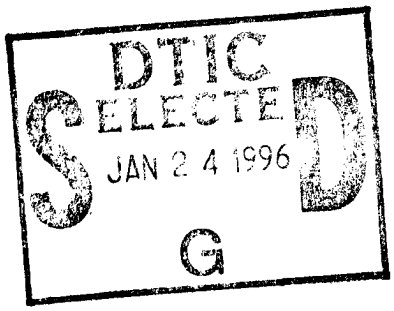
RL-TR-95-190
Final Technical Report
October 1995

A SOFTWARE DEVELOPMENT METHODOLOGY FOR PARALLEL PROCESSING SYSTEMS

University of Florida

Stephen S. Yau, Doc-Hwan Bae, Pranshu K. Gupta, Sun Il Paek, Thaddeus J. Thigpen, Jun Wane, and Michael A. Wells



APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Materiel Command
Griffiss Air Force Base, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-95-190 has been reviewed and is approved for publication.

APPROVED: Joseph P. Cavano

JOSEPH P. CAVANO
Project Engineer

FOR THE COMMANDER: John A. Graniero

JOHN A. GRANIERO
Chief Scientist
Command, Control & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL (C3CB) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	October 1995	Final Feb 93 - Sep 94

4. TITLE AND SUBTITLE

A SOFTWARE DEVELOPMENT METHODOLOGY FOR PARALLEL PROCESSING SYSTEMS

5. FUNDING NUMBERS

C - F30602-93-C-0054
PE - 62702F
PR - 5581
TA - 20
WU - PE

6. AUTHOR(S)

Stephen S. Yau, Doc-Hwan Bae, Pranshu K. Gupta, Sun Il Paek, Thaddeus J. Thigpen, Jun Wane, and Michael A. Wells

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of Florida
Computer and Information Sciences Department
Gainesville FL 33611-2024

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Rome Laboratory/C3CB
525 Brooks Rd
Rome NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

RL-TR-95-190

11. SUPPLEMENTARY NOTES

Rome Laboratory Project Engineer: Joseph P. Cavano/C3CB/(315) 330-4063

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

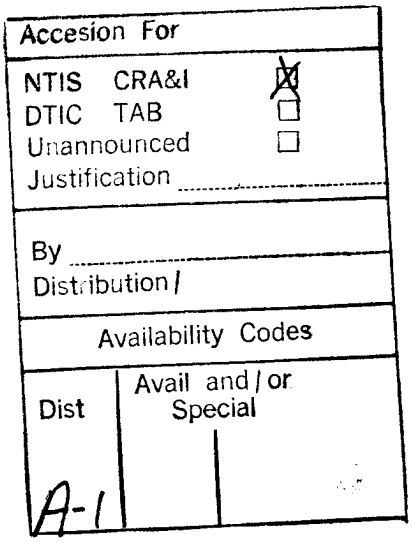
12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

A software development framework for parallel processing systems based on the parallel object-oriented functional computation model PROOF is evaluated. PROOF/L, a C++ based programming language with additional parallel constructs required by PROOF, is extended to include array data type and input/output features to make PROOF/L easier to use in developing software for parallel processing systems. The front-end translator from PROOF/L to the intermediate form IF1, and the back-end translators from IF1 to the C languages on two different MIMD parallel machines, nCube and KSR, are developed. Our framework is evaluated by comparing it with existing software development approaches for parallel processing systems. Our framework is suitable for large-scale parallel software development because it supports the concepts of hierarchical design and shared data, and frees the software developer from considering explicit synchronization, communication, and parallelism. The software development efforts using our framework can be greatly reduced due to implicit synchronization and communication and the compactness of PROOF/L programs. The extension of PROOF/L and the integration of PROOF/L with other programming languages to utilize existing library functions written in languages such as C and FORTRAN are also discussed.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Parallel software engineering, Parallel processing, Parallel software development	118
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL



Contents

List of Figures

List of Tables

Chapter 1

Introduction

In spite of rapid advances in computer technology, sophisticated Air Force applications, such as in the areas of C3I (command, control, communications and intelligence), avionics, logistics and engineering design, require great computing power to compute the required tasks within specified time frame. Such computing power often is not available in existing sequential computers. Parallel processing appears to be a promising solution to satisfy such needs. Currently, although there are various cost effective parallel processing systems available, effective utilization of such systems is still not achievable due to lack of effective methodologies to develop software for such systems. To fulfill this need, we have developed a software development framework [1, 2, 3] based on the computation model PROOF (PaRallel Object-Oriented Functional) [4] in which the object-oriented paradigm is integrated with the functional paradigm so that the software development framework has many useful features of both object-oriented paradigm and functional paradigm, such as understandability, reusability, extensibility, maintainability, expressiveness, and implicit parallelism. The major features of PROOF include expressing various granularity levels of parallelism, integrating referential transparency and history sensitivity, and supporting inheritance and synchronization without interference. Our software development framework is *architecture-independent* and thus can be used for developing software for various types of parallel processing systems. This software development framework covers from the requirements analysis and decomposition phase to the generation of target codes. In this framework, parallel aspects of the software are treated as a prime issue using object-oriented concepts by identifying parallel objects in the object decomposition phase of the software development. The programming language used in this framework is PROOF/L based on the PROOF computation model. PROOF/L is based on C++ with additional parallel constructs required in PROOF.

In this project, we have completed the following tasks:

- Development of a front-end translator from PROOF/L to IF1 [5], which is independent of the machine architecture. IF1 was chosen because it has been fully developed and can be used by a tool PAWS [6] for estimating the performance of the software to be developed.

- Development of two back-end translators from IF1 to the C languages run on two different MIMD machines: nCube and KSR. These translations depend on the machine architecture.
- Evaluation of the effectiveness of our software development framework for parallel processing systems.
- Extension of PROOF/L with the input/output functions and array construct.
- Investigation on the integration of PROOF/L with existing programming languages, such as C and FORTRAN.

In this report, we will briefly summarize in Chapter 2 existing parallel processing systems, and surveyed existing software development methods for parallel processing systems. In Chapter 3, our software development framework for parallel processing systems is summarized with an illustrative example. In Chapter 4, we will present the evaluation results of our software development framework by comparing it with other existing approaches, such as CODE(Computation-Oriented Display Environment) [7, 8], Proteus [9], Jagannathan's coarse-grain dataflow based methodology [10], Occamflow [11] and Pisa Parallel Processing Projects(P^4) [12]. In this chapter, we will discuss the effectiveness of our framework in terms of software development effort, application areas and the number of lines of code. In Chapters 5 and 6, the front-end translation process from PROOF/L to IF1 and the back-end translation process from IF1 to nCube C and from IF1 to KSR C will be presented. In Chapter 7, we will discuss extension of PROOF/L to include input/output features and array constructs. In Chapter 8, we will discuss our approach to integrating PROOF/L with existing languages, such as C and FORTRAN. In Chapter 9, we will compare IF1 with the Intermediate Program Representation IPR, which we used before [3], in terms of data dependency representation, application areas and maturity. Finally, the conclusions and future work will be given in Chapter 10. For the sake of completeness, we provide PROOF/L User's Manual and the syntax of PROOF/L as appendices.

Chapter 2

Background

In this chapter, we will briefly summarize the architectures of current parallel processing systems and software development methods available for these machines.

2.1 Parallel Computers

Flynn [13] classified computers into four categories based on the ways instructions and data are processed: SISD (Single Instruction stream, Single Data stream), SIMD (Single Instruction stream, Multiple Data stream), MISD (Multiple Instruction stream, Single Data stream) and MIMD (Multiple Instruction stream, Multiple Data stream). Among them, SISD computers are sequential computers which do not execute instructions or data in parallel, and commercial MISD machines do not exist. Thus, in this section, we will discuss SIMD and MIMD parallel machines.

2.1.1 SIMD Parallel Machines

In a SIMD parallel machine, processing elements are connected through an interconnection network, and are synchronously controlled by a central control unit. Each processing element has access to its own data, and thus the same operation can be performed simultaneously on many data items. Therefore, data parallelism is easily exploited among the processing elements in a SIMD parallel machine. The parallel machines belonging to this category include Connection Machine CM-2 [14], DAP (Distributed Array Processors) [15], and MasPar Mp-1 [16].

2.1.2 MIMD Parallel Machines

In comparison to SIMD parallel machines, an MIMD parallel machine consists of asynchronous parallel processors. Each processor has its own instruction and data set

to be processed. These processors communicate by passing messages among the processors. These machines can be further divided into shared memory parallel machines and distributed memory parallel machines [16].

An MIMD shared-memory parallel machine consists of a number of processors all having access to a single shared memory. The processors communicate by *read* and *write* operations, and are connected to the shared memory via one or more shared-buses or interconnection networks. As the number of processors in the system increases, the communication medium becomes a bottleneck in terms of performance as well as cost. Thus, a linear speed-up with an increase in the number of processors is not achievable or is limited to a certain number of processors. In MIMD shared-memory parallel machines, the major software design problems include data access synchronization and load balancing [16]. Shared-memory parallel machines include Cray X-MP and Y-MP series [17], Alliant FX8 [18], Encore Multimax [17], IBM RP3 [19], Sequent Balance [20], SGI PowerChallenge, and Convex Examplar.

An MIMD distributed parallel machine consists of a set of processors, each having a non-shared local memory. Processors communicate by message passing via communication channels. In MIMD distributed memory parallel machines, the synchronization is implicit through communication. The popular interconnection networks include *hypercube, ring, tree* and *mesh*. Unlike shared memory parallel machines, distributed memory parallel machines are not affected by the memory contention problem and are more easily expandable. One of the problems encountered by the distributed memory parallel computers is the message passing latency due to possible transferring of data via intermediate processors. The major software design problems include data placement, communication overhead and scheduling. The shared memory parallel machines can be considered as a special class of distributed memory parallel machines in which all the processors are fully connected [16]. Our software development approach based on PROOF [4] is targeted for MIMD distributed memory parallel machines. Any approach for MIMD distributed memory parallel machines can be adopted to MIMD shared memory parallel machines by modifying communication mechanism from 'message passing' to 'shared variable'. MIMD distributed memory parallel machines include hypercube [21], nCube [22], BBN [23], Inmos Transputer network [24], KSR [16], IBM SP-2, and DEC Alpha Server.

2.2 Software Development Approaches

Techniques for programming parallel processing systems can be classified in three categories [3]:

- Parallelizing or vectorizing compilers
- Parallel language constructs
- Parallel programming languages

Parallelizing or vectorizing compilers [26, 27, 28, 29, 30, 31] are widely used in conjunction with sequential programming languages, such as FORTRAN and C, usually for scientific computation. This approach is useful in that existing sequential software can be adapted to a parallel programming environment with minor modifications. However, parallelizing or vectorizing compilers can only detect parallelism associated with iterations over common data structures, such as arrays and matrices, and require extensive dependency analysis. Thus, it is not appropriate for developing large-scale software for parallel processing systems.

The parallel language constructs approach is to extend the existing sequential programming languages with parallel constructs, such as *input, output* constructs in CSP [32], *task* and *rendezvous* mechanisms in Ada [33], *in, out, rd, eval* constructs in Linda [34], *Mentat objects* in Mentat [35], and *collection, processors, distributions, alignment* mechanisms in pC++ [36, 37]. This approach requires programmers to explicitly specify the communication and synchronization among parallel processes. Thus, considering that many errors in parallel software stem from incorrect synchronization and communication, this approach may increase the software development effort.

Parallel programming languages are based on different paradigms. For examples, SISAL [38] is based on a functional paradigm tailored for scientific computation, PARLOG [39] is based on a logic paradigm, Act 1 [40] is based on an object-oriented paradigm, and PROOF/L [4] is based on object-oriented and functional paradigms. The underlying computation models of these programming languages are fundamentally different from those for imperative programming languages in that parallelism is mostly implicit and massive parallelism can be obtainable.

Our software development approach for parallel processing systems belongs to the parallel programming language category. It is based on the computation model PROOF in which an object-oriented paradigm is integrated to a functional paradigm in order to have the desirable properties of both paradigms. The object-oriented paradigm naturally reveals existing parallelism in the application problem structure [41]. Besides modifiability, maintainability and reusability, another advantage of PROOF is that the concept of an object can be used at earlier stages of the software development cycle than the implementation stage. Therefore, parallel processing aspects such as parallelism and communication among parallel components can be naturally handled at the earlier stages during software development. Consequently, it simplifies the handling of parallelism and communication among parallel components. However, in the object-oriented paradigm, parallel execution among concurrent objects is the only source of parallelism, and the amount of parallelism to be exploited may not be sufficient for effective utilization of fine-grain processors. On the other hand, the property of referential transparency obtained from functional languages based on the functional paradigm reduces programmers' efforts for dealing with explicit race conditions caused by multiple tasks. As a result, the functional paradigm has great potential to exploit implicit parallelism by removing side-effects caused by assignment statements. For this reason, functional paradigm has been studied as an alternative to the imperative programming languages for parallel programming [42].

We have compared our approach with the following existing software development approaches using parallel language constructs or parallel programming languages:

- CODE [7, 8]: a graph-based software development method for MIMD parallel computer systems
- Proteus [9]: a prototyping system consisting of a prototyping language and a transformation process to convert architecture-independent concurrent program to low-level code for the targeted parallel computer system
- Jagannathan's coarse grain dataflow-based methodology [10]: a dataflow-based methodology for coarse-grain multiprocessing on a network of workstations
- Occamflow [11]: a dataflow-based approach for programming multiprocessor systems on a network of transputers
- Pisa Parallel Processing Project(P^4) [12]: a programming method for general purpose distributed memory parallel processing systems.

These software development approaches for parallel processing systems focus on the exploitation of parallelism in numeric computation algorithms, and do not address large-scale software development issues, such as hierarchical structuring and data sharing. Except our approach, so far we have not seen any software development approaches for parallel processing systems which address such large-scale software development issues. Those approaches listed above and relevant parallel programming languages will be described and compared to our approach in Chapter 4.

Chapter 3

Our Approach

3.1 Overview of Our Approach

Our approach to software development for parallel processing systems is based on the computation model PROOF which incorporates the functional paradigm into the object-oriented paradigm.

Our framework, as shown in Figure 3.1, consists of the following phases: object-oriented analysis, object design, partitioning, PROOF/L coding, front-end translation from PROOF/L to IF1, grain size analysis, back-end translation from IF1 to a target language of a parallel processing system, and allocation. In the object-oriented analysis, the requirements are decomposed into a set of interacting objects. The concurrent/parallel aspects of the system behavior are analyzed and specified using the object-communication diagrams. The objects identified in the object-oriented analysis phase are then designed and verified in the object design phase. In the partitioning phase, the objects in the software system are partitioned into a set of clusters to improve the overall performance of the software system by minimizing communication cost and exploiting parallelism among objects. The front-end translation, grain size determination and back-end translation are grouped together by a dotted rectangle in Figure 3.1, called *transformation*, where the architecture-independent PROOF/L code is transformed into a target code to be allocated into the parallel machine. In the front-end translation, the PROOF/L code is translated into an IF1 code. In the back-end translation, the IF1 code is translated into the target code. The architecture dependent issues need not to be considered until after the front-end translation. In the grain-size analysis phase, the proper sizes of tasks are determined using the architecture-dependent information, such as communication cost and execution time in the target parallel machine. In this transformation, the partitioning and grain size analysis results are incorporated to generate the target code which can be efficiently executed on the target parallel machine. After the target code is generated, it is allocated to a set of processors. In the following sections, each phase in our approach will be summarized.

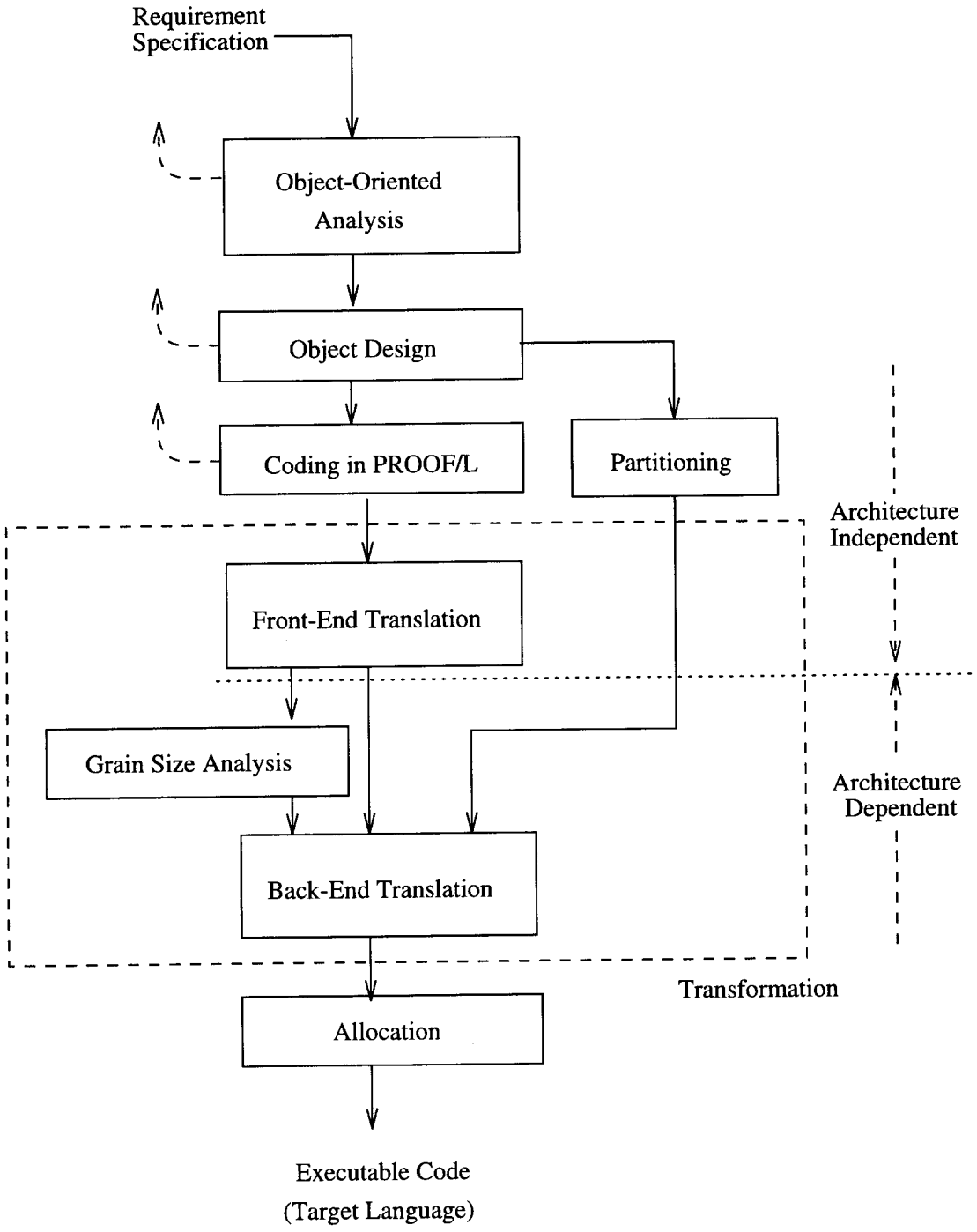


Figure 3.1: Our PROOF software development framework for parallel processing systems.

3.1.1 Object-Oriented Analysis

Our object-oriented analysis is different from other object-oriented analyses, such as Rumbaugh, et al's OMT(Object Modeling Technique) [43] and Coad & Yourdon's Object-Oriented Analysis [44], in that our approach focuses on the concurrent/parallel aspects of the system, but other approaches do not address concurrency explicitly. Our approach starts from the given requirement statements. The requirement statements often contain ambiguities. When ambiguities are found during the object-oriented analysis, we will report them to the user or domain expert to clarify the requirement statements. Thus, our object-oriented analysis is an iterative process which continues until all the functionalities are satisfactorily specified. When the requirement statements are not complete, we may use the guidelines given in [45] to clarify them although more research in this area is needed.

The object-oriented analysis phase consists of the following steps [1, 2, 3]:

1) Identify *objects* and *classes.*

2) Determine *class interfaces.*

3) Specify *dependency* and *communication relationships* among objects.

4) Identify *active*, *passive* and *pseudo-active* objects.

5) Identify the *shared* objects.

6) Specify the behavior of each of the objects.

7) Identify bottleneck objects, if any.

8) Check the completeness and consistency.

In Step 1), the software system is represented by a set of communicating objects. Objects are identified by analyzing the semantic contents of the requirement specifications. All physical and logical entities are recognized. Each object corresponds to a real-world entity, such as sensors, control devices, data and actions.

In Step 2), object class interfaces are determined. In PROOF, every object is considered as an instance of an object class. Instead of defining objects directly, the object classes to which they belong must be defined. Class interfaces may consist of both local methods and global methods. The local methods are class specfic methods; while global methods can be accessible to any other global methods, any method of other classes, or body of any object instance. The purpose of global methods is to provide a flexible way to address general operations which do not belong to any specific classes.

In Step 3), the static relationship among objects are specified using the object communication diagrams, in which the objects are represented as rectangles, the links between the objects (which can be specfied as method invocations) indicate the communication between objects, and the arrows on the links indicate the directions of communications.

In Step 4), the objects are classified according to their invocation properties as *active, passive* or *pseudo-active*. An active object can initiate activation of other objects by invoking methods of other objects. The methods defined in an active object cannot be invoked by other objects, but they can be invoked by other methods defined within the active object itself. A passive object is activated only when its methods are invoked by other objects. Pseudo-active objects can invoke the methods of other passive or pseudo-active objects and also has methods which can be invoked by other active or pseudo-active objects. All the threads of control in the application start from the active objects. We can identify all the possible threads of control and then use this information to check for the completeness and the consistency of the analysis.

In Step 5), once the static structure of the software system is determined, we identify shared objects from them. A shared object has local data which can be accessed by a number of objects. The shared objects can be further divided into *read-only* shared objects and *writable* shared objects. The read-only object has local data which cannot be modified by other objects. The writable object has local data which can be modified by other objects. Read-only objects can be freely duplicated as many times as desired. All the access to the data in the writable shared objects needs to be synchronized to maintain the consistent status of the data.

In Step 6), the behavior of each object is specified using the following notations:
- SEQ$(m_1, m_2, \ldots, m_n)$: The methods $m_1, m_2, \ldots, m_n$ are executed sequentially.
- CON$(m_1, m_2, \ldots, m_n)$: The methods $m_1, m_2, \ldots, m_n$ are executed concurrently.
- WAIT(m, O): Object is waiting for the invocation of its method m by another object O to proceed with its execution.
- SEL$(m_1, m_2, \ldots, m_n)$: The object selects one of the methods for execution from among the methods $m_1, m_2, \ldots, m_n$.
- ONE $-$ OF$(WAIT(m_1, O_j), \ldots, WAIT(m_n, O_k))$: The object permits only one of its methods $m_1, \ldots, m_n$, to be invoked by other objects. ONE-OF construct is used in cases where other objects could try to invoke the methods defined in the object O simultaneously, while the object O permits only one object to invoke its method at a time.

In Step 7), bottleneck objects which may unnecessarily degrade the performance of the software system are identified. Usually, a bottleneck object will be a shared writable object. One can identify a shared writable object from the description of the object behavior in Step 6). If such an object is found, then redo or refine the analysis to reduce the bottleneck if possible.

In Step 8), the result of the analysis is verified with the user requirements. From the given user requirements, the possible threads of controls are identified, and each of them is examined using the behavior of the objects specified in Step 5).

For more detailed information on the object-oriented analysis in our approach, refer to [3].

3.1.2 Object Design

Objects obtained from the analysis phase have to be designed. In our approach, the object design is specified using the notation defined in PROOF/L [4]. The class interface definitions and information about the object behavior are used to design the objects. Our approach to object design involves the following four steps:

1) Establish the class hierarchy.

2) Design the class composition and the methods in each object.

3) Design the bodies of the active and pseudo-active objects.

4) Verify the object design.

In Step 1), since some common operations and/or attributes between the objects may not be apparent in the analysis phase, different objects are reexamined to identify the commonality between the classes in the design phase. A set of operations and/or attributes that are common to more than one class can then be abstracted and implemented in a common class called the *superclass*. The subclasses then have only the specialized features.

In Step 2), the composition and the methods for each object class are designed. The class definition consists of *composition* and *methods*. The composition defines the internal data structure of the class. Various constructors, such as list and Cartesian product, are provided. A typical functional style is adopted in the method definition. A rich set of functional forms, i.e. high-order functions, as well as primitive functions are predefined. In the method design, the internal state of the object to which the method belongs is included as both the input and output parameters so that side-effects are avoided. A method of an object consists of an optional *guard* and an *expression*. The guard is a predicate specifying synchronization constraints and the expression statement specifies the behavior of the method. The object which invokes the method is suspended when the value of the attached guard is `False`, and it is resumed when the guard becomes `True`. The guard attached to a method is defined in a way that it only depends on the status of the local data, and does not depend on the definition of any other methods. The global methods which are class-wide methods should also be specified here for determining the properties of the operations.

In Step 3), a body is associated with each active and pseudo-active object. There is no body associated with a passive object as it does not invoke any methods. The role of a body is to invoke a method and to modify the state of the objects represented by their local data. The body in each object is expressed in the form $e_1//e_2//\ldots//e_k$ where each e_i is an expression representing method invocations and expressions separated by // are evaluated simultaneously. // is a parallel construct indicating parallel execution. The modification of an object is expressed by the *reception* construct which has the form $R[|o|]e$, where o, called a *recipient object*, is an object name and e is an expression with applications of purely applicative functions only. The reception

construct can occur only in the bodies of active and pseudo-active objects. The reception construct indicates that the object o will receive the value returned as a result of evaluating the expression e. This construct modifies the states of the object.

In Step 4), the design of the objects done in the previous phase has to be verified and analyzed. For this purpose, we transform our design into Petri nets [46], which have been selected in our approach mainly because our design can be easily represented in a Petri net model and because many techniques have been developed to analyze Petri-net models. The transformation of our design to Petri nets consists of the following three steps: 1) transformation of bodies to Petri nets, 2) composition of the nets, and 3) refinement of the nets.

For more detailed information on the object-oriented design steps in our approach, refer to [3].

3.1.3 Partitioning

In the partitioning step, the objects in the software systems are partitioned into a set of clusters in order to reduce communication cost among processors while maintaining the parallelism among the objects. It is very difficult to achieve linear speedup due to communication costs among processors, contention of shared resources and inability to keep all the processors busy [47]. That is one of the reasons that there is a large gap between the ideal peak performance and the real performance in most parallel computers. The partitioning approaches for reducing communication cost are divided into three categories: graph-theoretic [48, 49], integer programming [50, 51] and heuristics [52]-[56]. One of the common assumptions in these approaches is that the execution time for each module and the communication time among modules are given as input. Our partitioning approach does not assume that exact execution time and communication time are available. In addition, most of the existing partitioning approaches cannot be used when the software is decomposed as a set of such objects having shared data.

The objective of our partitioning approach is to improve the overall performance of the software by reducing communication cost among processors while maintaining the potential parallelism among objects. The input to our approach are (1) the behavior of the objects in the software system, expressed using the constructs, such as SEQ, CON, WAIT, SEL and ONE-OF, (2) communication information extracted from the requirements analysis, and (3) the number of replications for each object as required for fault tolerance. Using this information, we represent the software system as an undirected weighted graph in which every node represents a cluster of objects and every edge between two nodes has a weight representing the degree of contribution for improving the overall performance of the software system by parallel execution of the two clusters. The details of our partitioning approach with illustrative examples has been presented in [57].

3.1.4 Transformation

The transformation of the PROOF/L code to a target code involves the following steps: partitioning, front-end translation, grain-size determination and back-end translation.

The PROOF/L code is first translated into an IF1 code and then the IF1 code is translated to the target code. The former is called *front-end* translation which is a semantics-oriented translation, and machine or architecture dependent issues are not involved. The latter is called *back-end* translation.

In the grain size analysis step, we focus on finding proper grain sizes within each object. Thus, we can consider each object as an independent program. We represent the program as a directed graph in which each node corresponds to an IF1 construct, and each edge represents a data dependency relation. In order to perform grain size analysis, the execution time of the IF1 constructs is estimated statically, and the communication time between them are estimated by examining the type information of the data transmitted. The estimation can be done statically by analyzing the assembly code for these constructs. We developed efficient heuristic algorithms of three different types of parallelism – tree parallelism, graph parallelism and pipe-lined parallelism. The details of these algorithms can be found in [3].

The back-end translation is performance-oriented, and machine or architecture dependent parameters, such as communication types, number of links and number of processors are used to perform various analyses. After partitioning and grain size analysis information is incorporated to the intermediate form, the intermediate form is translated into corresponding equivalent target code In this project, we have developed two back-end translators for two target parallel processing systems, KSR and nCube. However, current implementation does not include partitioning and grain size determination. The details of the front-end and back-end translations will be discussed later.

3.1.5 Allocation

After the target code is generated, the target code is allocated to the parallel processors in such a way that the execution time of the target code can be minimized by exploiting parallelism in the target code.

One of the problems that must be solved in order to achieve high performance of software for parallel computers is the allocation of tasks among the processors. Some of the factors that prevent the ideal linear speed-up in parallel processing are 1) insufficient concurrency and 2) high communication overhead [58]. The task allocation problem has been studied extensively [58]-[61]. In these approaches, efficient heuristic task allocation algorithms were introduced. Factors to be considered in the allocation phase include the number of processors, the number of processes to be allocated, interprocessor communication pattern, and communication overhead.

3.2 An Example

In this section, we will use a hypothetical example to demonstrate our approach involving both coarse grain and fine grain parallelism. We will use the same hypethetical example described in our previous report [3]: An air force base defense system consisting of three air force bases. In that example, communication and synchronization aspects among air force bases were emphasized (coarse-grain parallelism). Here we will focus on one of the bases and emphasize both communication and computation aspects. The specfication, analysis, design, and coding processes of this example, the same as in [3], are included here for the sake of completeness.

3.2.1 Specifications of a Hypothetical Air Force Base Defense System

Assume that there are three air force bases that are closely connected. For the sake of simplicity, we assume that only one type of flighters, one type of bombers, one type of surface-to-air missile batteries for defensive purposes against the attacking enemy. Radars and C3 (Command, Control and Communication) facilities are available. Each base may have many radars, but the base gets only one correlated radar value. Each base will also have several missile batteries and sufficient missiles to be used for its defense. Each base has either fighters or bombers for the defense. There would be one central C3I unit which advices each base as to what it should do for its defense purposes. In our application, we will associate the C3I advice for a base along with the design of the base itself since this is a parallel processing system. This way, the commander at the center can know what is going on at different bases simultaneously and will also be able to give orders to the different bases simultaneously.

For the example shown below, we will characterize one of the bases and emphasize more computation aspect on that base. The detail description is as follows:

An air force base consists of radar installations, equipment, and armed personnel. The radar detects approaching hostile attacks on the base. It is assumed that the enemy cluster consists of either bombers or missiles, but not a combination of the two. The base, in turn, can use its fighters or its missiles, but not a combination of the two simultaneously, to defend itself from these attacks. The defense strategy used by the base depends on the configuration of the enemy cluster.

Upon detection of an enemy cluster, the radar tracks the cluster to determine its composition. This enemy information, which is the number of bombers or missiles of each enemy cluster, is stored in a queue. The air force base retrieves the enemy information from the queue.

If the enemy cluster consists of x bombers, the base defends itself by launching either its fighters [represented by the computation of the function $F(x)$] or its missiles [represented by the computation of the function $G(x)$]. On the other hand, if the enemy cluster consists of y missiles, the base defends itself by launching its own missiles to intercept the incoming missiles [represented by the computation of the function $H(y)$].

In our implementation, we use a random number generator to simulate various incoming threats. For simplicity purpose, we assume that

$$F(x) = \sum_{i=1}^{x} i,\ i \epsilon I \tag{3.1}$$

$$G(x) = \pi \simeq \frac{1}{x} \sum_{i=1}^{x} \frac{4.0}{1.0 + {\chi_i}^2},\ \chi_i = \frac{(i - 0.5)}{x},\ i \epsilon I, \tag{3.2}$$

$$H(y) = \{z_1, z_2, z_3, ...\},\ z_i \epsilon \{Prime\ numbers\},\ 1 \leq z_i \leq y \tag{3.3}$$

Each of the defense strategies [the computation of $F(x)$, $G(x)$ and $H(y)$], and the radar will be executed in parallel on independent nodes to exploit *coarse grain parallelism.* Threats are added to a FIFO queue. The air force base removes a threat from the FIFO queue and computes either $F(x)$, $G(x)$ or $H(y)$ depending on the type of the threat.

Each of the defense strategies is executed on multiple processors to exploit *fine grain parallelism.* It does so by breaking down its task into smaller tasks which can then be executed in parallel on independent nodes. The results of these smaller tasks are then gathered together to yield the final result.

3.2.2 Object-Oriented Analysis

Identifying Classes and Objects

We identify the following *Classes* from the requirements specification of the example:

- Base – for air force base
- Radar – for radar associated with Base.
- Queue – for FIFO queue.

From the requirements specification, we identify the following objects:

- B – corresponding to a single air force base.
- R – corresponding to the radar associated with the single air force base.
- Q – for recording enemy cluster information.

Defining Class Interfaces

Class interface of an object consists of the input and output parameters and their types. Shown below are the class interfaces of the various objects identified in the previous subsection. As an example, let us consider the class *Queue*. We show one interface which is called method *Insert*. This method is invoked by the body of object *R*. From the domain knowledge of the example, we can infer that the radar value consists of the number of bombers or missiles attacking the base. The type of data is obviously to be integer and is the same as below. Thus, in a similar fashion, the class interfaces for the various classes can be determined. In order to illustrate the usage of global methods, the class interfaces for the classes *Base* and *Radar* consists of global methods, not class specific methods. The reason for using the global methods is that the computation, such as random number generator, finding prime number, π approximation, factorial summation, are all general operations which do not belong to any specific class. All class interfaces of this example are given below:

```
class Queue

   method Q_init(s:int -> Queue)

   method Insert ( -> Queue)

   method Assign (New:list -> Queue)

   method Delete ( -> Queue)

   method GetElem( -> int)

end class

class Base

   method IsPrime(number:int, factor:int -> int)
   method Prime(number:int -> int)
   method FindPrimes(low:int, upp:int, number:int -> int)

   method IntegerSum(l:list -> int)
   method FactSum(low:int, upp:int -> int)

   method RealSum(l:list -> real)
   method Pi(l:int, h:int, interval:int -> real)

end class


class Radar
```

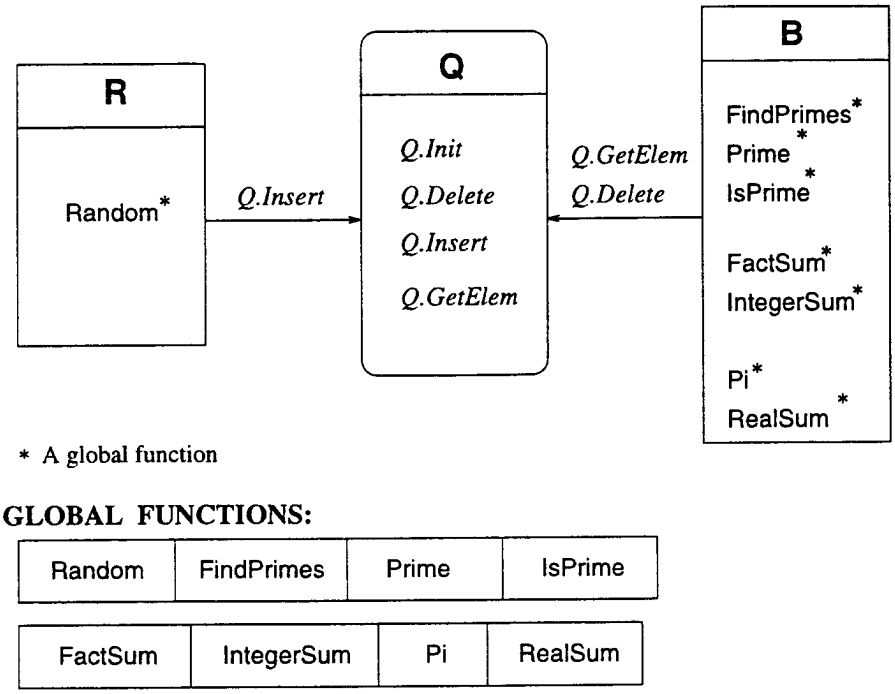


Figure 3.2: The object communication diagram for the set of decomposed objects of the hypothetical air force base defense example.

```
   method Random(low:int, upp:int, number:int -> int)

end class
```

where Random,IsPrime, Prime, FindPrimes, IntegerSum, FactorialSum, RealSum, and Pi are global methods and do not belong to any specific class.

Specifying Dependency and Communication Relationships Among Objects

Once the class interfaces are obtained for all the classes, we can establish the dependency and communication relationships among the objects from the object-oriented analysis phase. Figure 3.2 gives the dependency and communication relationships among these objects. To illustrate the operation, let us consider the object R, which puts a radar value into the object Q. Thus, there exists communication between R and Q.

Identifying Active, Passive and Pseudo-Active Objects

From the requirements specification and from the object communication diagram shown in Figure 3.2, we can see that the object R is not invoked by other objects, but does invoke other objects such as Q. Thus, R is identified as an active object. To

Table 3.1: Object classification of the hypothetical air force base defense example.

Classification	Objects
Active	R, B
Passive	Q
Pseudo-active	None

illustrate the methods for identifying passive objects, let us consider the communication behavior of the object Q, which is invoked by other objects such as R and B, but never invokes any other objects. Such objects are classified as passive objects. If the communication behavior shows an object being invoked by other objects as well as invoking other objects, it is identified as pseudo-active object. Figure 3.2 shows no such object. Thus, we have no pseudo-active objects in this example. We can classify the objects as shown in Table 3.1.

Identifying Shared Objects

From the object communication diagram as well as the object behavior, we identify the object Q as a shared writable object. Shared writable objects are usually passive objects.

Specifying the Behavior of Each Object

We are now in a position to describe the behavior of each object. For instance, let us consider the object R. The object R adds threats to a FIFO queue endlessly. Thus, we have the behavior of the object R. The behavior of each object is given below:

Behavior of object R:

```
while(TRUE,
      (Q.Insert))
```

Behavior of object B:

```
while(TRUE,
      let low = 1 in
      let upp = 10001 in
      let third = (/ (- upp low) 3) in
      let target = (Q.GetElem) in
         Q.Delete,
         if ( (null? target),
         # THEN do nothing because no threat to base
```

```
    # ELSE respond to threats
        if ( (<= target third),
            # THEN-clause
            (FindPrimes, 1, target)
            # ELSE-clause
            if ( (and (> target third) (<= target (* 2 third)) ),
                # THEN-clause
                (FactorialSum, 0, target),
                # ELSE-clause
                (Pi, 1, target, (- target 1))
))))
```

Identifying Bottleneck Objects

We have identified the object Q as shared writable object. Since different methods of this object are used by the objects R and B to access this object, the access to the object Q does not have to be serialized. Hence, Q is not a bottleneck object. Thus, we do not have any bottleneck objects in this example.

Checking for Completeness and Consistency of the Object-Oriented Analysis

Because of the similicity of the example, by tracing through the behavior of the objects and looking at the class interfaces, we can see that the object-oriented analysis is complete and consistent.

3.2.3 Object Design

Establishing Hierarchy

Since in this example we do not have two different types of objects with some common behavior, we do not need to define a superclass. In other words, we do not have any inheritance in this example.

Designing Class Composition and Methods

The class composition typically consists of local data present in the class. The type of data present in the class is also identified. In this stage, we also provide the methods present in each of the classes. As an example, consider the class composition of the class *Queue*. The data in the object is a list of integers generated by the global method *Random* and two integers. These constitute the class composition. In addition to these, we define the methods. The methods required for the class *Queue* are as follows:

1. Initialize the integers in the composition.
2. Add integers to the list *TargetsQ* in the composition.
3. Modify the list *TargetsQ* in the composition.
4. Delete integers from the list *TargetsQ* in the composition.

These are defined formally as follow:

```
Class Queue

   composition
      TargetsQ :  list
      seed     :  int
      number   :  int
   end composition


   method Q_init(s:int -> Queue)
      expression
         object Queue (seed = s, number = 0)


   method Insert ( -> Queue)
      expression
         let low = 1 in
         let upp = 10001 in
         let item = (Random low upp seed) in
         object Queue ( TargetsQ = (append_right TargetsQ item), seed = item )


   method Assign (New:list -> Queue)
      expression
         object Queue ( TargetsQ = New )


   method Delete ( -> Queue)
      #guard (> number 0)
      expression
         object Queue ( TargetsQ = (tail TargetsQ), seed = seed )


   method GetElem( -> int)
      #guard (> number 0)
      expression
         (head TargetsQ)
```

```
end class


global

   method Random (low:int, upp:int, number:int -> int)
      expression
         let factor = (- (/ (+ low upp) 2) 13) in
         let x = (mod (* factor number) upp) in
         if ( (and (and (>= x low) (<= x upp)) (> x 0)),
              (+ x 0),
              (Random low upp x))


   method IsPrime(number:int, factor:int -> int)
      expression
          if ( (<= (* factor factor) number),
               if ( (= (mod number factor) 0),
                   0,
                   (IsPrime number (inc factor)) ),
               1 )


   method Prime(number:int -> int)
      expression
         if ( (or (= number 2) (= number 3)),
              1,
              if ( (= (IsPrime number 2) 1),
                   1,
                   0 ))


   method FindPrimes(low:int, number:int -> list)
      expression
         (head while (lambda (x) (> (head (tail x)) (head (tail (tail x))) ),
                      lambda (x) (
                         let y = (head (tail x)) in
                         let z = (head (tail (tail x))) in
                         if ( (= (Prime y) 1),
                              [ (append_right (head x) y) (- y 1) z ],
                              [ (head x) (- y 1) z ] ))
                      ) [ [] number low ] )


   method IntegerSum(l:list -> int)
```

```
      expression
         if ( (null? l),
              0,
              (+ (head l) (IntegerSum (tail l))) )


   method FactSum(low:int, upp : int -> int)
      expression
         (head while (lambda(x) (< (head (tail x)) (head (tail (tail x)))),
                      lambda(x)
                         let y = (head (tail (tail x))) in
                         [ (+ (head x) y) (head (tail x)) (- y 1) ]
                         ) [ 0 low upp ] )


   method RealSum(l:list -> real)
      expression
         if ( (null? l),
              0,
              (+ (head l) (RealSum (tail l))) )


   method Pi(l : int, h : int, interval : int -> real)
      expression
         (head while(lambda(x) (< (head (tail x)) (head (tail (tail x)))),
                     lambda(x)
                        let w = (/ 1.0 (head (tail (tail (tail x))))) in
                        let t = (* (- (head (tail x)) 0.5) w) in
                        let tmp = (/ 4.0 (+ 1.0 (* t t))) in
                        [ (+ tmp (head x)) (+ 1 (head (tail x)))
                          (head (tail (tail x))) (head (tail (tail (tail x)))) ]
                        ) [ 0 l h interval ] )

end global
```

Designing the Body of the Objects

The body of an object describes the control thread within the body. A control thread exists for only active and pseudo-active objects. Thus, the bodies exist for only active and pseudo-active objects. In our example, the bodies exist for the objects R and B since these objects have been identified previously as active objects. The behavior of the active objects should describe the body of that object. For example, the object R has a body which iteratively executes in accordance with its behavior specified before. In the following, we give the body of each active object.

Body of object R:

```
while(TRUE,
      ;(
        R[| Q |] (Q.Insert),
        (delay 1)
        ))
```

Body of object B:

```
while(TRUE,
      ;(
        let low = 1 in
        let upp = 10001 in
        let third = (/ (- upp low) 3) in
        let target = (Q.GetElem) in
        ;(
           R[| Q |] (Q.Delete),
           if ( (null? target),
                # do nothing because no threat to base
                (delay 2),
                ;(
                  # respond to threats
                  if ( (<= target third),
                       ;( R[| B |] object Base (result =
                                  (delta (FindPrimes, 1, target))) ),
                          if ( (and (> target third) (<= target (* 2 third)) ),
                               ;( R[| B |] object Base (result = [(IntegerSum
                                        (delta (FactSum, 0, target)))]) ),
                               ;( R[| B |] object Base (result = (/ (RealSum
                          (delta (Pi, 1, target, (- target 1) ))) (- target 1) )) )
                          ))
                ))
       )))
```

Verification

In the first step, the bodies of the active and pseudo-active objects are transformed into Petri nets. The transformation of the bodies of the objects in this application are shown in Figures 3.3 and 3.4.

The second step is to examine these nets to reduce the number of independent Petri nets. The nets are composed at the fusion point, also called the bottleneck place so that shared modifiable objects are serialized for access among different objects. For example, the object Q is a shared writable object that is modified by the objects R and B. Thus all the transitions in Figures 3.3 and 3.4 corresponding to the methods in Q are to be fused together at the bottleneck place. This process of fusing will bring

the different nets together.

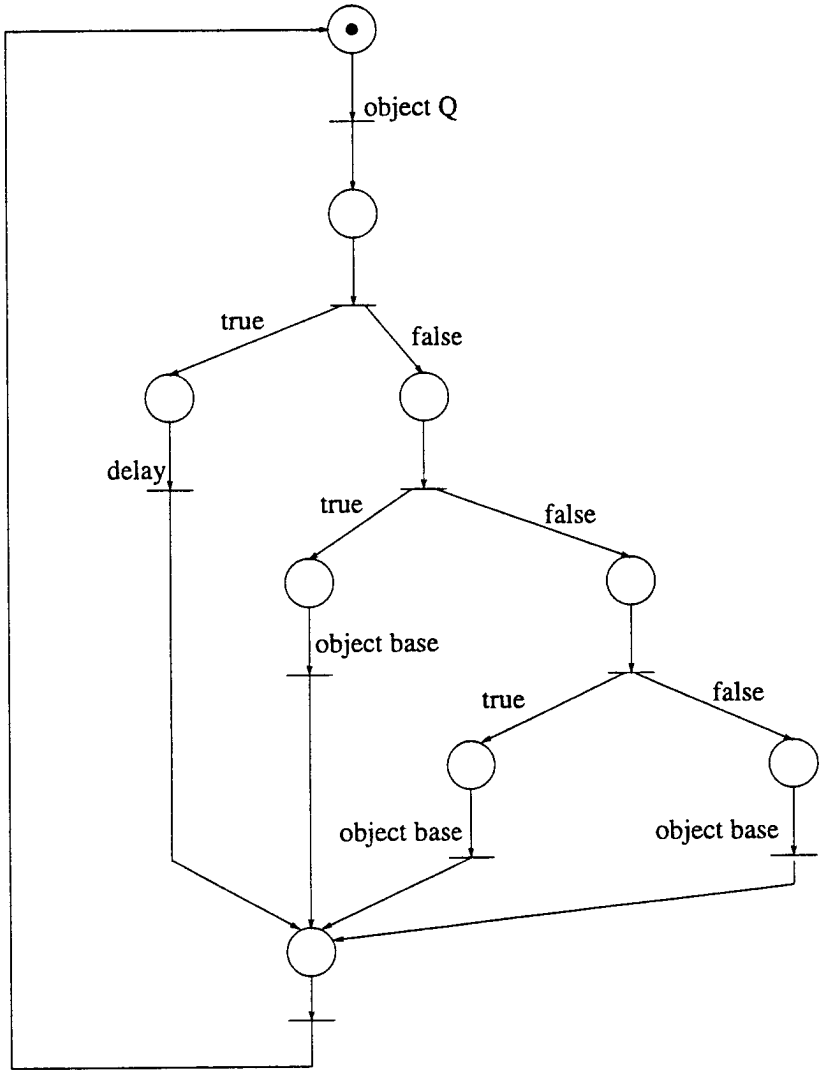


Figure 3.3: Transformation of object B, an instance of class *Base*, to a Petri net

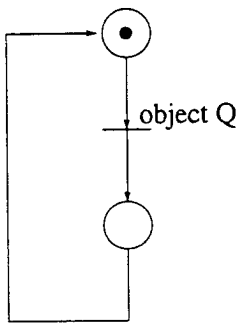


Figure 3.4: Transformation of object R, an instance of class *Radar*, to a Petri net

The last step is to refine the above nets to reflect the details of each method. This is achieved by expanding each transition to show the guard andthe expression. This has been illustrated earlier in our framework. Once the Petri nets are obtained, we can then apply the available techniques to verify that the Petri nets satisfy the necessary properties. The detail method for verification of object design is referred to [3].

Chapter 4

Evaluation of Our Approach

In this chapter, we will compare our approach with other existing approaches, and evaluate the effectiveness of our approach in terms of software development effort. We will present the lines of code comparison results between PROOF/L programs and their equivalent nCube C and KSR C programs in the hypothetical air force base defense system example described in Chapter 3. We will also present the speed-ups obtained in the π approximation problem.

4.1 Comparison of our approach with other parallel software development approaches

In this section, we will survey other existing parallel software development approaches, such as CODE [7, 8], Proteus [9], Jagannathan's dataflow-based approach [10], Occamflow [11] and Pisa Parallel Processing Project(P^4) [12], and compare them with our approach in terms of parallelism supported, underlying computation model, scope, target machine and application areas as shown in Figure 4.1.

	CODE	Protues	Jagannathan	Occamflow	P^4	PROOF
Parallelism	data function	data function	function	function	data function	data function
Computation Model	graph-based object-oriented	functional	demand-driven	data-flow	conventional	object-oriented funtional
Scope	Design Implementation	Design Implementation	design	implemetation	implementation	design implementation
Target Machine	general	general	workstation clusters	transputers	general	general

Figure 4.1: Comparison of parallel software development approaches

At the end of this section, we will briefly summarize the parallel programming lan-

guages relevant to PROOF/L.

4.1.1 Computation-Oriented Display Environment(CODE)

The Computation-Oriented Display Environment(CODE) is based on the premise that parallel programming systems should be based on a well-defined, comprehensive parallel computation model rather than merely a collection of implementation models [7]. In CODE, a parallel program is represented by a set of *computation-units* and *dependency units*. The software development method based on CODE consists of the following steps:

1) Draw the program's dependency graph, in which each node represents a computation-unit and an arrow between nodes represents a dependency relation.
2) Define the dependencies among the computation units by completing specification forms.
3) Complete the definition of the computation units.
4) Specify the firing rules, which indicate the state of the computation unit to start execution.
5) Create a standard, architecture-independent program specification.
6) Map the dependencies and firing rules into parallel-computation structures.

In [8], CODE has been extended with data partitioning functions and integrated with an object-oriented paradigm. This approach is similar to our approach in that an architecture-independent specification is designed and then it is mapped into the architecture-dependent program. The goal of their work is to build a graph-based development environment for software development for parallel processing systems. However, the underlying computation model does not support the concepts of shared data and exploiting parallelism in various granularity levels.

4.1.2 Proteus

The Proteus [9] is a prototyping system to develop software for parallel processing systems. It consists of the following steps:

1) Specify architecture-independent concurrent program using Proteus language.
2) Evaluate using Proteus interpreter and measurement tools.
3) Refine expression of concurrency in Proteus program to target execution on particular parallel platform.
4) Transform Proteus program to low-level code for targeted parallel platform.

5) Execute program using native compilers and supporting libraries.

In the Proteus system, the data parallel operations are expressed using the mathematical notations of set, sequence and map comprehension. The function parallelism is expressed with a small set of process creation and synchronization primitives, and communication is based on the shared variable. This approach is similar to our approach in that the architecture-independent high-level specification is transformed into an architecture-dependent target code to be executed in parallel. It has an advantage over our approach in that it can evaluate the design by prototyping with the Proteus interpreter. However, this approach is not suitable for large-scale software development for distributed or parallel processing systems due to lack of hierarchical design concept. It does not explicitly supports abstraction mechanism or constructs to allow expressing several layers of design hierarchically, which is enssential for developing large-scale software.

4.1.3 Jagannathan's Coarse-Grain Dataflow based Methodology

Jagannathan, et al [10] developed a dataflow-based methodology for coarse-grain multiprocessing on a network of workstations. In this approach, an application is expressed as a data-dependency graph in which vertices are function modules and whose edges represent data dependencies between function modules. Each function is described in conventional code where the input parameters of the module correspond to the incoming edges of the associated vertex and the output parameters of the modules correspond to the outgoing edges of the vertex. They also developed a language, a directed graph language (DGL) to express coarse-grained parallelism in applications. DGL is similar to other directed-graph-based languages, but it differs in the granularity of a basic operations. The granularity of a basic model in this approach is coarse. The model supports a declarative paradigm based on dataflow among the modules, and a procedural paradigm inside modules. The execution system embodies the tagged demand-driven execution model, and provides the following functions:

- Process demand for a data value from a function module.
- Allocate the most suitable workstations to compute a demanded data value.
- Schedule execution of a vertex when appropriate data values are available.
- Communicate demand for a data value and the resulting data value between workstations.
- Manage shared data value.
- Provide the programmer with an interface to run the application.

It implements each vertex of a program as a process and each edge as a communication channel on which messages can be exchanged using interprocess communication mechanisms. This approach does not include the design steps, but supports only graphical user interface for the programmers to specify the high-level design.

4.1.4 Occamflow

Gaudiot and Lee [11] developed a methodology for programming multiprocessor systems based on data-flow model of computation. Although the authors called Occamflow a methodology, this is a set of translation steps from a SISAL program to an Occam program rather than a design methodology. The input to this methodology is a SISAL program and the output of the methodology is an equivalent Occam program. This approach consists of the following steps:

1) Translate the SISAL program into IF1.

2) Scan the IF1 and generates a graph which consists of two subgraphs: the Program Structure Graph(PSG) and the Data-Flow Graph(DFG).

3) Generates a Partitioned Data-Flow Graph(PDFG), a channel table and a communication cost matrix.

4) Optimize through repartitioning.

5) Generate the Occam program.

The translation steps from SISAL to Occam through IF1 are similar to our approach. However, this method does not include design steps and support the shared data concept. In addition, the parallelism can only be exploited through the Occam programming environment.

4.1.5 Pisa Parallel Processing Project(P^4)

In the Pisa Parallel Processing Project(P^4) [12], an approach to program general purpose distributed memory parallel processing systems is developed. The P^4 approach is based on two major components: a high-level programming language, the Pisa Parallel Processing Language (P^3L), and an abstract Parallel Memory (P^3M). The P^3L language allows the programmer to explicitly express parallelism in an application at a high level. The P^3L language includes the following parallel constructs to express various types of parallelism:

- pipeline: models process pipelines acting on streams of input data, and sequential execution of processes.
- farm: models different farms, having workers executing either the same function on different data (function replication) or different functions (function partitioning).
- tree: models computations having either a static tree or a dynamic tree structure in a way similar to demand-driven or divide-and-conquer models of execution.
- loop: adds a feedback channel from the last to the first process of a particular construct in a way which is similar to data-driven modeling of iterative programs.

The P^3M is based on the idea that a general-purpose parallel machine must achieve the balance between local and nonlocal communication. The P^3M supports efficient mapping to different physical parallel computers. In this approach, two issues, the *locality* in process communication and *dynamicity* in the memory management and process structure, are dealt with to implement high-performance applications on parallel processing systems. When an application is mapped onto P^3M, the programming tools decide whether a process communication is to be implemented through local communication or through nonlocal communication. Although the author claims this approach is a design methodology, this approach is focused on how to map the application program into physical processors with distributed memory rather than a software design method for parallel processing systems.

4.1.6 Others

In this section, we will compare programming languages which were developed as implementation languages for software development for parallel processing systems.

Linda [34] is a small set of operations that can be added to a base language to create a parallel processing dialect. The concept of Linda is based on the tuple space of parallel processing. Processes and data can be considered to be elements in tuple space. Communication between processes occurs in the following way: the sender creates data in tuple space, and the receiver gets the data in the tuple space, and thus communication takes place. Linda provides the following four basic operations: *in*, *out*, *rd* and *eval*. *in* removes the tuple which was read from the tuple space, *rd* reads the tuple, but leaves the tuple to be read by other processes, *out* creates a new tuple and places it in the tuple space, and *eval* creates a new tuple by generating a process. A disadvantage of Linda is that the programmers have to write programs in terms of communication with other processes. In addition, its implementation on distributed memory parallel processors requires significant overhead to support communication via shared memory. When we consider the fact that data or messages need not be accessible to processes other than those processes which need them, Linda cannot support an information hiding principle.

Goldberg [62] developed a method to programming parallel processors for functional programs by introducing a logical construct called a *serial combinator*. A serial combinator is defined as a function with the following properties: 1) its body contains no free variables; 2) its body is sequential and contains constructs for synchronizing its execution with other tasks; 3) its body could not occur as a subexpression within the body of another serial combinator. In this approach, the third property implies that the programmers have to determine as few serial combinators as possible, since they cannot be coalesced to form a bigger serial combinator. It also implies that the program developed for one parallel computer may not be directly portable to another due to possible performance degradation. In addition, the programmers must ensure the correct synchronization and communication among tasks.

Foster [63] introduced *Strand* for parallel programming, which is based on logic com-

putation model. Strand can provide an interface to other languages as in [10]. He does not consider the granularity of parallelism and assume that the programmers will make choices on the grain size during the development of the application program.

Grimshaw [35] developed an object-oriented programming language Mentat for parallel processing systems by extending C++ with parallel constructs. The Mentat programmer makes granularity and encapsulation decisions, and the compiler manages communication and synchronization. The underlying computation model supports parallelism among the objects, and is based on a medium-grain data driven model in which programs are directed graphs. The vertices of the program graphs are computation elements, and the edges denotes the data dependencies between the vertices. This approach can exploit task or function parallelism, but is not appropriate to exploit massive data parallelism.

Gannon and Lee [36] also developed a parallel object-oriented programming language for parallel processing systems by extending C++ with parallel constructs, such as *collection, processors, distributions* and *alignments*. A collection can be an array, a grid, a tree or any other partitionable data structure. Processors are objects that are used to build distributions for collections each of which represents a set of threads of control. Distribution and alignments are the mechanisms to assign distributed elements onto the processors. These extensions allow the programmer to express data parallelism easily, but are not suitable to express and exploit task or function parallelism.

4.2 Software Development Effort

To demonstrate the effectiveness of our approach, we have compared the effort required to develop software directly using the C of two MIMD machines, nCube and KSR, with that required to develop the same software in PROOF/L using our approach. We measured the time needed to complete the development of the software for the air force base defense example described in Chapter 3.

For this example, when we used the nCube C language directly, we spent approximately 25 hours on coordinating the communication among different processors and setting up communication channels and buffers, besides the computational portion. The nCube C programs involve explicit communication and synchronization handling and pointer manipulations. On the other hand, when we used PROOF/L, the same example took us only 5 to 6 hours to complete. Although it may be too early to draw a conclusion based on our experience in this particular example, we believe that the advantage of implicit communication and synchronization in PROOF/L significantly contributed to the reduction of software development effort.

4.2.1 Architecture Independence and Translator Development Effort

Usually parallel programs are not portable to different parallel machines. PROOF/L, being an architecture-independent parallel programming language, shields all the underlying machine-dependent details from programmers and can run on various parallel machines as long as the machines have the PROOF/L back-end translators. The back-end translators of PROOF/L have been implemented on the distributed memory nCube and the shared memory KSR parallel machines. Machine-dependent details are incorporated in the back-end translation process.

Our experience has shown that after we implemented the translator for one parallel machine, it was much easier for us to implement the translator for other parallel machines. Most of the code that involves common computation has close similarities, except the parts involving communication and synchronization which highly depend on the architectures of parallel machines. For instance, after we had implemented the PROOF/L back-end translator for nCube C which took us approximately 150 hours, we spent only about 40 hours to implement the PROOF/L back-end translator for KSR C. We anticipate that if we implemented the PROOF/L back-end translator for KSR C first, it should also take about 150 hours.

Besides parallel machines, we are also interested in implementing PROOF/L on distributed workstation cluster systems and extending it to various distributed computing systems, like autonomous decentralized systems [64, 65].

4.2.2 Implicit Communication and Synchronization

As mentioned before, one of the major reasons that the software development effort required using our approach is considerably less than that when we develop the software directly using the target languages is that our approach supports implicit communication and synchronization. PROOF/L objects are loosely coupled, and interactions among objects are realized through method invocations of other objects, which are similar to normal functional calls. Our current prototype does not incorporate global method invocations, which will be implemented in the near future. Guard structures embedded in method bodies are for synchronization among objects. The bodies of methods can only be executed when guards are true; otherwise methods will be blocked until guards become true. Guards are evaluated based on the object's local information and arguments passed to the methods. The following two examples, which are extracted from the air force base defense example described in Section 3.2, will show how we realize communication and synchronization through guard statements and method invocations:

Example 1. Guard structure: The following code segment shows that the guard statement in the method ensures that the *queue* object for storing enemy information, which is the number of bombers or missiles of each enemy cluster, is not empty when the base wants to extract such information from the *queue*.

```
class Queue

composition
number: int
...

method GetElem( -> int)
guard (> number 0)
expression
...
(head TargetsQ)

...

end class
```

Example 2. Method invocation: The following code segment shows that whenever the *radar* object detects a threat, it will invoke the method *insert* of the *queue* object for recording the enemy information.

```
Body of object R:
loop(
      ;(
         ...

         (out 'Radar detects a threat'),
         R[| Q |] (Q.Insert),
         ...

       )
    )
```

4.3 Application Areas

Our approach exploits coarse grain parallelism by deriving all the concurrent objects from a problem and classifying them into different categories: active, pseudo active or passive. Our approach can easily be applied to general communication-oriented problems in which a number of objects need to be executed simultaneously, and these objects interact with one another periodically. It has been applied to software development for distributed computing systems [2]. The air force base defense example in

Chapter 3 is a communication-oriented and computation-oriented application. The synchronization among different objects, such as the radars and the bases, has been realized by using guard structures within object methods as illustrated in Section 4.2. Method invocations fulfill communication among different objects.

Our approach also exploits fine grain parallelisms at the method level. Parallel functions specify data or functional parallelisms in a method. It is suitable for computation-oriented applications, such as the air force base defense example, where we embed all the computations inside object methods and distribute these computations on different nodes. The comparison of the execution time and the speedup for the air force base defense example and computing π using various numbers of nodes of the nCube are shown in Tables 4.1 - 4.4 and Figure 4.2.

Table 4.1: The execution time of the hypothetical air force base defense example programmed directly in nCube C using different numbers of nodes.

Num of Nodes Used	Air Force base defense (sec)
4	639.89
8	417.47
16	254.26
32	150.67
64	115.05

Table 4.2: The execution time of the hypothetical air force base defense example programmed in PROOF/L and then translated to nCube C using different numbers of nodes.

Num of Nodes Used	Air Force base defense (sec)
4	1135.68
8	592.25
16	385.73
32	314.15
64	240.98

Tables 4.1 and 4.2 show the execution time for the air force base defense example in 100 iterations (the program itself goes infinitely) using nCube C directly and using PROOF/L run on the nCube through the PROOF/L back-end translator. All the

Table 4.3: Comparison of the execution time (in micro seconds) of the programs for computing π based on (3.2) which are generated by directly programming in nCube C.

Interval	1	2	4	8	16	32	64
10^4	$5.53 * 10^4$	$2.80 * 10^4$	$1.45 * 10^4$	$0.82 * 10^4$	$0.60 * 10^4$	$0.68 * 10^4$	$1.14 * 10^4$
10^5	$5.53 * 10^5$	$2.77 * 10^5$	$1.39 * 10^5$	$7.04 * 10^4$	$3.71 * 10^4$	$2.23 * 10^4$	$1.93 * 10^4$
10^6	$5.52 * 10^6$	$2.76 * 10^6$	$1.38 * 10^6$	$6.92 * 10^5$	$3.48 * 10^5$	$1.78 * 10^5$	$9.70 * 10^4$

Table 4.4: Comparison of the execution time (in micro seconds) of the programs for computing π based on (3.2) which are generated by programming in PROOF/L and then translated to nCube C.

Interval	1	2	4	8	16	32	64
10^4	$2.08 * 10^5$	$1.39 * 10^5$	$7.54 * 10^4$	$4.39 * 10^4$	$2.90 * 10^4$	$2.28 * 10^4$	$2.07 * 10^4$
10^5	$2.62 * 10^6$	$1.32 * 10^6$	$6.63 * 10^5$	$3.38 * 10^5$	$1.76 * 10^5$	$9.63 * 10^4$	$5.89 * 10^4$
10^6	$8.70 * 10^6$	$8.90 * 10^6$	$6.55 * 10^6$	$3.28 * 10^6$	$1.65 * 10^6$	$8.33 * 10^5$	$4.23 * 10^5$

time scale is in seconds. The speedup of more nodes used to execute the program generated by directly programming in nCube C is about the same as that generated by programming in PROOF/L. On the other hand, the execution time for executing the program generated by directly programming in nCube C is always smaller than that generated by programming in PROOF/L, but much more effort is needed for direct programming in nCube C than that for programing in PROOF/L.

The execution time (in micro seconds) for the programs computing π generated by direct programming in nCube C and generated by programming in PROOF/L is presented in Tables 4.3 and 4.4. The formula for computing π is given in (3.2). The corresponding speed-up curves are shown in Figure 4.2, and they indicate better speed-up and approach the ideal speed-up when the data interval is increasing.

Noted that the exection time for PROOF/L code shown in Table 4.4 is not as good as that for nCube C shown in Table 4.3. We believe that some work, such as incorporating object-partitioning algorithms into the PROOF/L translation process [3], would help improve the performance of the translated PROOF/L programs substantially.

4.4 Lines of Code Comparisons with Target Languages

In the Figure 4.3, we present line comparisons among several programs in PROOF/L, native nCube and KSR C (direct programming), translated nCube and KSR C. These programs are π approximation, the air force base defense system example presented in our previous report [3] (air force base defense example part I), and the air force base defense example described in Chapter 3 (air force base defense example part II).

According to the results shown in Figure 4.3, PROOF/L code has the advantage over direct coding in C and translated C code from PROOF/L in terms of the number of lines of code, except that the air force base defense example part I in which there is considerable duplication of code because we have not fully implemented the inheritance in PROOF/L.

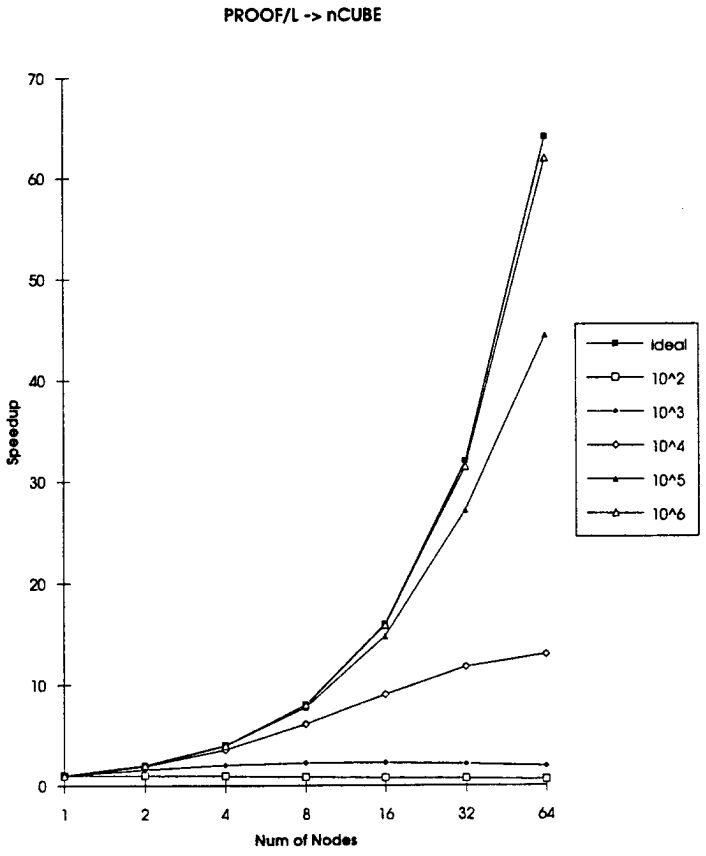


Figure 4.2: Speedup using various numbers of nodes of nCube to compute the π in PROOF/L.

Program \ Language	PROOF/L	Direct Programming		Translated from PROOF/L	
		nCUBE C	KSR C	nCUBE C	KSR C
π Approximation	15	79	61	400	350
Air Force Base Defense Example Part I	700	500	400	3000	2000
Air Force Base Defense Example Part II	156	411	760	1651	1179

Figure 4.3: Comparison of the number of lines of code for programming in C directly and in C translated from PROOF/L.

Chapter 5

PROOF/L Front-end Translation

As mentioned in Section 3.1.4, the translation of PROOF/L to a target code of a parallel machine involves partitioning, front-end translation, grain-size determination, and back-end translation. In this chapter, we will focus on the front-end translation, which is independent of the architecture of the parallel machine used. The front-end translator transforms PROOF/L code to a superset of IF1, which is a dataflow language based on acyclic graphs. We will briefly discuss the IF1 language, and then the translation process from PROOF/L to our superset of IF1. Special attention is paid to the extensions and modifications we have made to IF1, which are necessary to support PROOF/L. Finally, we will describe the architecture of the front-end compiler, which uses the UNIX tools *lex* and *yacc*. The syntax and semantics of PROOF/L are discussed in Appendix A PROOF/L Reference Manual.

5.1 IF1

IF1 is a text representation of acyclic dataflow graphs. It was originally developed as the intermediate language for SISAL, a high-level functional programming language [38]. IF1 was adopted for use with the PAWS project [6], a parallel systems performance analysis tool which we had hoped to use with PROOF/L.

There are four types of entities in IF1: nodes, edges, types, graph boundaries. Nodes represent operations, either logical or mathematical. A node can be either *simple* or *compound*. Simple nodes represent basic operations such as addition, absolute value and equality operations (<=, > , etc.). Compound nodes contain subgraphs; these subgraphs may contain other IF1 nodes. Compound nodes are used for selection and looping operations. A node has input and output ports. Edges are used to represent the data flow of a program. Edges connect the output ports of nodes to the input ports of nodes. Edges and node ports in IF1 are associated with a type. There are six basic types in IF1 (boolean, character, double, integer, null and real). From these

types, complex types may be derived (such as records, streams and tuples).

An IF1 program may consist of many different graphs. These graphs are delineated by graph boundaries. For more detailed information of IF1, the reader is referred to [5].

5.2 Translating PROOF/L constructs to IF1

5.2.1 PROOF/L Types

The BOOLEAN, INT, REAL and CHAR types of PROOF/L all map to equivalent types in IF1. The STRING type is only used with the `out` debugging function, and uses the same IF1 type as CHAR. At present, arrays are not supported in this implementation.

The composition of a PROOF/L class is represented using the record type of IF1.

5.2.2 The `list` Type

PROOF/L supports lists. A list is an ordered group of (possibly) heterogeneous elements. There are six basic types provided by IF1: boolean, character, double, integer, null, and real. From these types, more complex types may be created: records consisting of a fixed number of fields of various types, an array of a particular type, etc.

The PROOF/L `list` type, however, cannot be expressed using pure IF1. IF1 records are of a fixed size; PROOF/L lists can be any size. IF1 arrays contain elements of a single type; PROOF/L lists can contain elements of many types. Therefore, a new basic type is introduced in our superset of IF1 – the list.

Lists are flexible, but support for heterogeneous lists requires significantly more time and space overhead than other less flexible structures. Furthermore, compile time error detection becomes more difficult when lists are introduced. The advantages of a strongly typed language cannot be realized with non-strongly typed constructs like lists.

5.2.3 Class Declarations

A class consists of a composition and a set of methods. The class composition is treated as an IF1 record. An IF1 graph is produced for each method in the class.

IF1 does not have direct support for object-oriented languages. The class name is added to the method name when defining IF1 graphs that correspond to PROOF/L methods.

5.2.4 Function Calls

A function call uses IF1's **CALL** node. The current implementation of PROOF/L does not support polymorphism. For IF1 to support polymorphism, it is necessary for the program at runtime to route function call requests to the appropriate method.

Example:

```
class A
   method Test(->int)
      expression
        (+ GetValue GetValue)

   method GetValue(->int)
      expression 1
end class

class B of A
   method GetValue(->int)
      expression 2
end class

class C of A
   method GetValue(->int)
      expression 3
end class
```

In the above code fragment, classes B and C are subclasses of A. Classes B and C inherit the `Test` method and override the `GetValue` of class A.

When a call is made to `Test`, the `Test` method must route the calls to `GetValue` of the appropriate class. Therefore, a call to `GetValue` by instances of A, B and C should return the values 2, 4 and 6, respectively. Currently, there is no method of conveying this information in IF1.

Alpha and Beta Functions

The alpha and beta functions of PROOF/L are syntactic sugar – an alpha or beta function may be rewritten as a series of function calls. Therefore, when the front-end translates an alpha or beta function to IF1, it converts the function to the equivalent form.

5.3 New IF1 Nodes

Some of the features of the PROOF/L language necessitate the addition of new nodes to the superset of IF1. We have added the following new nodes: RECEPT, LBUILD, RGET, and GUARD.

5.3.1 Reception Pseudo-Function

PROOF/L supports persistence with the *Reception Pseudo-Function.* RECEPT accepts two arguments – the name of the object to be modified and the new value of the object and returns the new value of the object, as shown in Figure 5.1.

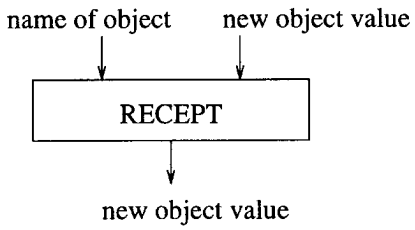


Figure 5.1: The IF1 node RECEPT.

5.3.2 List Construction

As mentioned above, the `list` type in PROOF/L is not directly supported by IF1. Similarly, there is no IF1 node to create a list. Therefore, we added a new node to IF1 to support list construction called `LBUILD`. LBuild accepts n inputs of any type and returns a list, as shown in Figure 5.2.

5.3.3 Retrieving the Value of an Object

A PROOF/L method may refer to the composition of an object. Since the composition of an object may be modified using the reception function, a special node was added

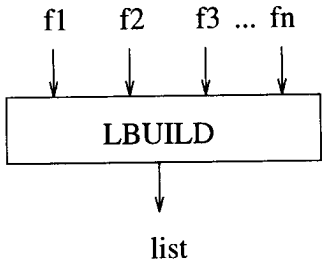


Figure 5.2: The IF1 node LBUILD.

to IF1 to fetch the current value of an object called `RGET`. RGET accepts a single string literal argument which corresponds to the name of the object, and returns a record containing the object's composition, as shown in Figure 5.3.

Since there may be numerous instances of class objects, RGET accepts the special keyword `self`, which refers to the instance of the object calling a class method.

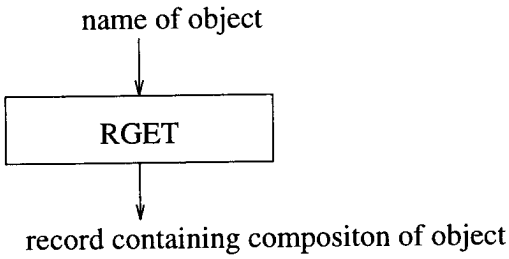


Figure 5.3: The IF1 node RGET.

5.3.4 Guards

Synchronization among objects is achieved by attaching an optional precondition, or *guard* expression, to class methods. The object which invokes the method is suspended when the attached guard expression becomes `False`, and resumed when the guard becomes `True`.

Although a busy waiting implementation of the guard could be expressed in IF1 using the existing looping constructs, we decided to add a new compound node to IF1. By using a special node for guard, we do not have to specify how the guard will be implemented, and we maintain the property of architecture independence at this stage. It is shown in Figure 5.4.

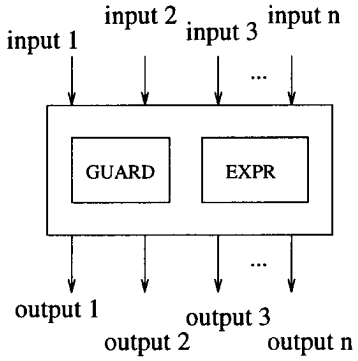


Figure 5.4: The IF1 node GUARD.

5.4 Implementation of the PROOF/L Front-End Translator

The front-end translator consists of three main modules: scanner, parser and symbol table. The scanner and parser were created with the aid of the UNIX tools *lex* and *yacc*, respectively, as shown in Figure 5.5. This is an extension of the implementation reported in [3]. We expand IF1, as shown in Section 5.3, for accommdation of better representations of PROOF/L structures.

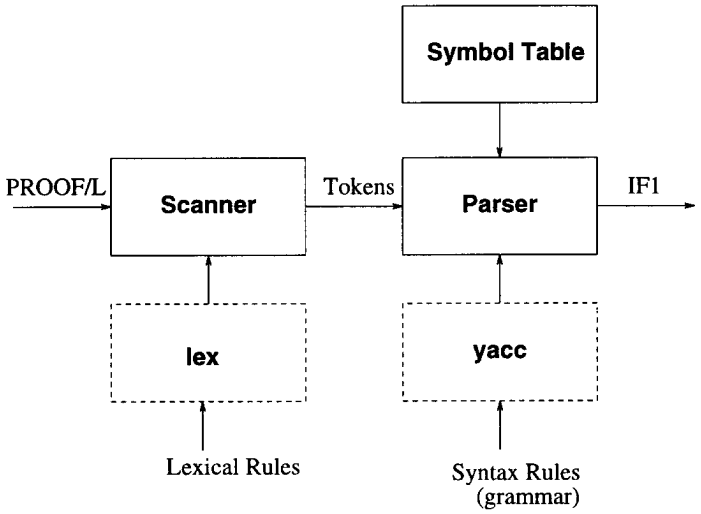


Figure 5.5: Architecture of the PROOF/L front-end translator.

5.4.1 Lexical Analyzer

The function of a lexical analyzer is to group the input character stream into a token stream and as input of the latter parser phase. A token is a basic element of parsing.

In this part, we use the UNIX tool *lex* to generate the code of the lexical analyzer. We specify the lexical rules in regular expressions in the *lex* language; lex compiles the

input file and generates a C program which simulates the corresponding finite state machines needed for lexical analysis.

The input format of *lex* is divided into three parts:

<definitions>

%%

<rules>

%%

<programmer subroutines>

The first part <definitions> and third part <programmer subroutines> are optional.

In the first part *definitions*, we can specify some sets of the lexical rules in the next part. For example,

```
letter          [a-zA-Z]
digit           [0-9]
letter_or_digit [a-zA-Z_0-9]
sign            [+-]
```

In the second part <rules>, we can use these defined sets to express the lexical rules. For example, the lexical rules for integer numbers:

```
digit+  {
            yylval.y_int = atoi(yytext);
            return token(INTEGER);
        }
```

The left-hand side part is the regular expression of an integer and the right-hand side part is the corresponding actions of an integer token: converts the text into the number and return a token INTEGER.

The last part <programmer subroutines> consists of some C routines written by users.

Another part of this module is the screener. The function of the screener is to distinguish key words from identifiers; because both are the same in structure, they cannot

be efficiently distinguished by regular expressions. The screener checks an identifier against a sorted keyword table using binary search; if the identifier is found in the keyword list, the appropriate token is returned. Otherwise, an identifier token is returned.

5.4.2 Parser

The parser checks the correctness of input and generates IF1 code. The parser is was produced using the UNIX tool *yacc* .

The input format of *yacc* is similar to that of *lex*. It also consists of the same three parts (definitions, rules and programmer subroutines) and they are also separated by two "%%". Only the second part is compulsory and the other two are optional.

The first part of the parser is the definitions. We need to give the definitions of tokens, return types, priority between operators and start rule of the grammar. For example,

```
%token          INTEGER
        ...
%type   <y_int> INTEGER
        ...
%start  proofl
```

The second part of the parser is the most important part, including grammar rules and corresponding actions. For example,

```
proofl  :  PROGRAM ID COLON class_list obj_list body_list END
{
           body();
}
;
```

This is the starting grammar rule of a PROOF/L program. It begins with the reserved word **program** and the name of this program. After a ";", the rest of the program is the class declarations and object declarations. Finally, it is the list of bodies of active objects and ended with the reserved word **end**. Similar to *lex*, between a pair of "{" and "}" is the corresponding action part of this rule. For the example above, the action is calling the function body() to build the object body list.

In our implementation of the front-end translator, the parser generates the IF1 code as it parses a PROOF/L program. The grammar rules for PROOF/L are given in Appendex B.

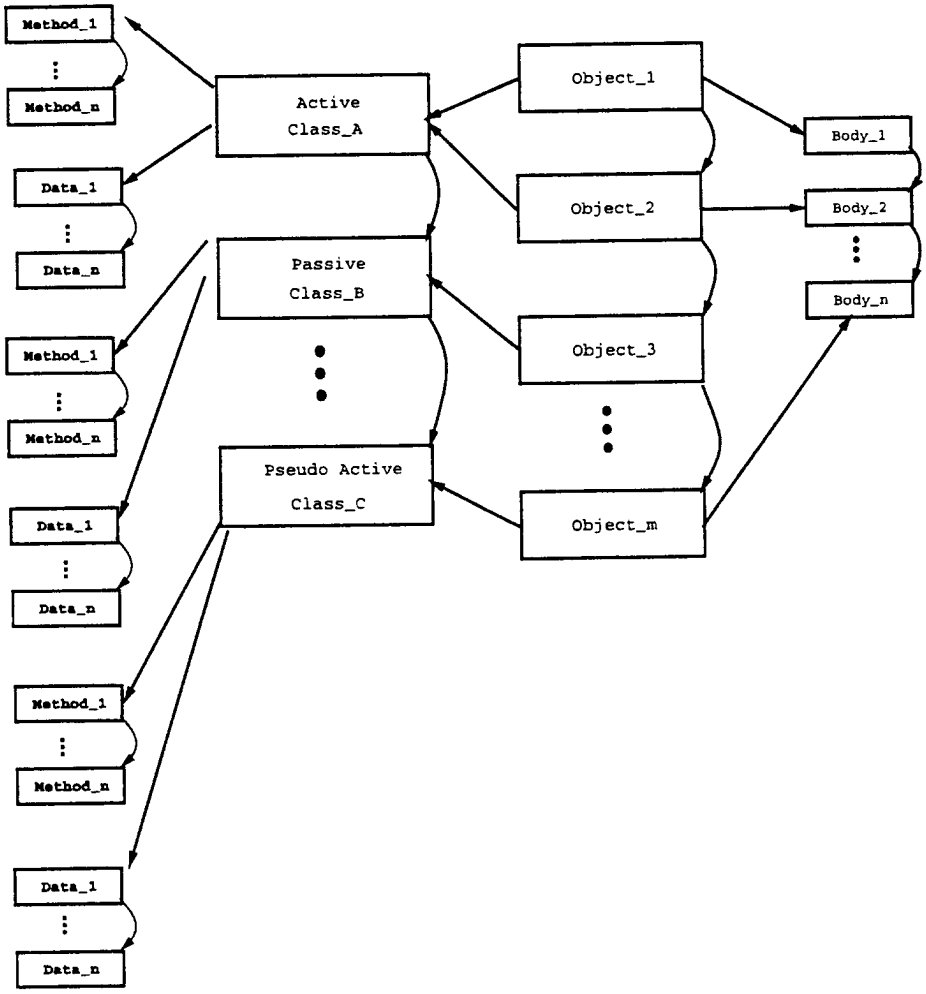


Figure 5.6: The structure of the symbol table of the PROOF/L front-end translator.

5.4.3 Symbol Table Handling

The symbol table used by the parser stores information about classes, methods, objects and the composition of objects.

As class, method and object declarations are parsed, appropriate entries are made to the symbol table. The information stored in the symbol table is used to verify the validity of the PROOF/L program and to generate IF1 code.

The relations among the symbol tables are shown in Figure 5.6.

Chapter 6

PROOF/L Back-end Translation

In this Chapter, we will present the back-end translation which translates our superset Intermediate Form 1 (IF1) to two parallel dialect C languages: nCube C and KSR C. Various IF1 constructs for parallel functions, iterative WHILE loops, IF structures, and common computational operations are identified and translated. The important issues about designing inter-node communications and synchronizations are also discussed.

6.1 Target Languages

As mentioned before, there are two kinds of MIMD parallel architectures: shared-memory and distributed-memory. In the shared-memory architecture, the processors share single memory resource. In the distributed-memory architecture, each processor has its own memory, cooperative work must be done through explicitly specified inter-node communications and synchronizations.

We have developed two back-end translators: one for a distributed-memory parallel machine nCube and the other for a shared-memory parallel machine KSR.

6.1.1 The nCube C

The nCube C version 3 [66] consists of a comprehensive set of ordinary C primitives and build-in functions. Several basic primitives for the parallel execution and inter-node communication are:

- **rexec:** launch an executable program on a subset of processors. It involves allocation of a set of processors, setup of a process table for each processor within

the node set, and execution of the program on each processor.

- **ntest:** a non-blocking way to test existence of messages from other nodes.
- **nread:** waits for messages and reads them whenever they arrive and satisfy the type format set by the nread. The nread operation is self-blocked. Inappropriate nread operations could lead to deadlocks.
- **nwrite:** sends messages from one node across the nCube high-speed bus to another one with a type format.

All these primitives have susbstantially large communication overheads.

There are several other functions used to check states of processors at the run time on the nCube parallel machine:

- **whoami:** reports a node condition during the run time.
- **npid:** return current node ID.
- **ncubesize:** return the hypercube size, which is 2's power.

The nCube C itself does not provide any primitives to prevent deadlocks or to synchronize physical nodes. Each node basically stands alone itself.

6.1.2 KSR C

KSR C version 1.0.3[67] not only fully supports ordinary C primitives and built-in functions, but also supplies a comprehensive set of primitives for parallel execution, barrier synchronization, mutual exclusion and monitor. The shared-memory architecture implement all the details about interactions among different processors. The primitives used in the translation are:

- **pthread_create:** create a thread which shares the single memory.
- **pthread_join:** a master pthread waits for terminations of slave pthreads.
- **pthread_barrier:** master/slave type control to synchronize all the slaves to start simultaneously.

The KSR kernel handles all the communications among different threads distributed on physical nodes.

In general, ways to optimize communications among all the nodes involve partition, allocation, and grain size determination, which currently have not been implemented in the translator.

6.2 From IF1 to Target Languages

As mentioned in Chapter 5, IF1 (Intermediate Form 1) is used to have graphical representations for PROOF/L programs.

The translation from IF1 to target languages consists of three steps: parse IF1 code, structural linking and translation, as shown in Figure 6.1.

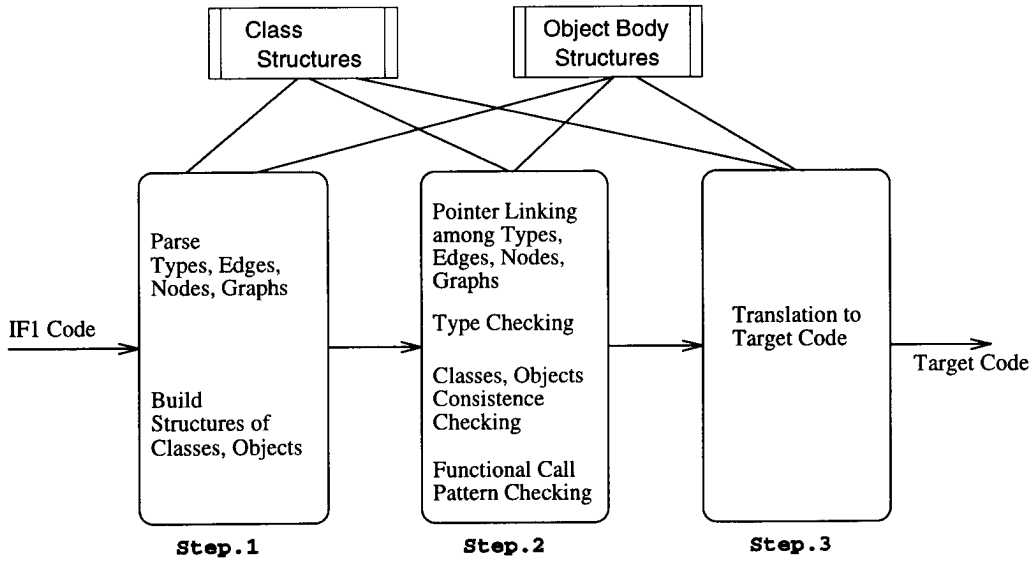


Figure 6.1: Translation steps from IF1 to a target code.

6.2.1 Parse IF1 Code

IF1 code consists of types, edges, nodes, graphs, and numerical relations among them. Because there are no explicit mechanisms to describe object-oriented concepts – classes and objects – in the original IF1 syntax, extensions have been made for the IF1 code to keep the class and object information for the back-end translation. All the class and object information is stored in the IF1 type headers, graph headers, and object headers. The following IF1 code describes these constructs.

1. The class's composition:

   T <Type_id> RECORD<next> %na =<class_name>

2. The class's method header:

   G <Type_id> <Class_Name.Method_Name>

3. The object body part:

X 0 <Class_Name.Object_Name>

The comment fields are only used for reference and ignored during the translation.

The execution sequence within a graph is sorted by detecting the dependency among the nodes, edges, and their numerical linkages in the graph. The algorithm below demonstrates the sorting of execution sequence:

```
seq = 0;
mark all the nodes unsorted;
for (each node) {
  for (each input edge) {
     if (each input edge is literal ||
              not an output edge from an unsorted  node) {
        continue;
     } else {
       break;
     }
  }
  if (all the input edges are checked) {
    set order of current node = seq;
    seq++;
  }
}
if (any unsorted nodes left) {
  report error;
  exit(-1);
}
```

After *types*, *edges*, *nodes*, and *graphs* of the entire IF1 code have been scanned, class and object structures can be derived through implicit class and object information in the new nodes of the IF1 code introduced in Section 5.3. The information includes class compositions in *types*, method names in *graphs* and object names in *graphs*.

6.2.2 Structural Linking

After numerical relations among types, edges, nodes, and graphs have been identified and built in the data structure during the first step, all these numerical relations are converted to pointer linking between types and edges, edges and nodes, types and graphs, and nodes and graphs. Method calls include four types:

- Built-in functional calls (denoted as imported functions in the IF1 code). They include `append_left`, `append_right`, `tail`, `head`, `last`, `inc`, `dec`, `null?`,

`delay`, `while`, etc.

- Global functions. The class GLOBAL is a class without any data structure but methods. It is a collection for public methods, in which every method can be used by any other classes or object bodies without any difference comparing to using their own methods.
- Method calls within a class. Currently a class method only can call another method within the scope of the same class, besides build-in and global functions.
- Method calls within an object body. Objects can call any methods available within the entire scope.

In addition, GUARD structures within methods are detected for each class. It is the only way for different objects in the PROOF/L code to synchronize one another. These structures are represented as GUARD compound nodes introduced in Section 5.3. The text representation is shown as follows:

```
{
G       0       GUARD 1 (structure)
C The predicate needed to be realized in order to continue the execution
...
G       0       GUARD 2 (structure)
C The body to be executed after the guard predicate above becomes true.
...
} <node_id>     GUARD    2    1    2
```

The number of input edges to computational nodes are verified, and edges for input arguments to method CALL nodes are checked against method prototypes. Type consistency checking is also applied to input and output data flows, which are represented by edges, among all the simple nodes in the IF1 code.

PROOF/L parallel structures will be detected in two ways at the IF1 level:

- Detecting `alpha` (apply to all) and `beta` (distributed apply) parallel function forms through data dependency among CALL and LBUILD nodes.
- find a built-in parallel function call named `delta` (data partition) in the IF1 code.

6.2.3 Translation

In order to realize the functional features in the PROOF/L language, only two kinds of data formats have been used: ATOM and SEXP. They are shown in Figure 6.2.

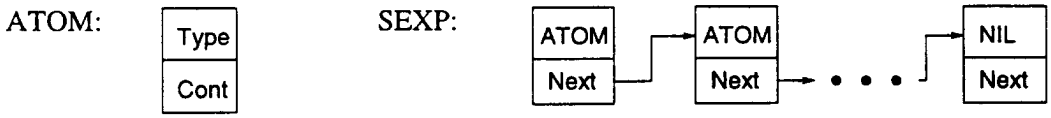


Figure 6.2: The data format of the translated target C code

Concatenated structures for PROOF/L class compositions are represented by using SEXPs. An initialization function for each class will be provided to initialize all components within the correspondent class composition: 0 for the integer or float, FALSE for the boolean, ''(empty string) for the string, NIL for the list. So there are no explicitly data definitions of classes at the C code level.

Our own PROOF/L library routines for supporting functional operations have been provided. They include all the built-in functions (except parallel `delat` function), copy routines, and garbage collections routines. Besides all these routines, packet assembling and disassembling routines have been written for message-passing type communications in the nCube C code, which will be described later.

In order to modularize the entire translated target C code, four basic files are generated after the translation:

- **class.h** contains all the object declarations and all the necessary C "include" files. All class method prototypes and class initialization function prototypes are also listed here. This is the main header file for the entire translated C code.
- **methods.c** contains all the class methods, bounded by comment marks for each class. It also contains class-method lookup tables for the purpose of communications among different objects, which will be further explained.
- **objects.c** contains bodies of all the objects.
- **main.c** provides the initialization of all the available physical nodes, and associate each object body with a single node in the nCube C (create a thread for each object body in the KSR C), dispatch all the correspondent controls to object bodies, synchronize all the objects to start execution simultaneously, and finally do the cleanup when all the objects are terminated.

The broad translation for a class is described as follows:

```
IF1                                    nCube C
C Class <name>                         /* Class <name> begins */

C Class Composition                    /* Class composition */
...

C Class Methods                        /* Methods <Class_name.Method_name 1>
```

```
                                          begins */
G <Type_id>  <Class_name.Method_name 1> void <Class_name.Method_name 1>
                                               (<I/O Type>)  {
...                                       ...
                                          }  /* End of method */
...                                       ...

C Class Methods                           /* Method <Class_name.Method_name k>
                                             beings */
G  <Type_id> <Class_name.Method_name k> void <Class_name.Method_name k>
                                               (<I/O Type>)  {
...                                       ...
                                          }  /* End of method */
end Class                                 /* Class ends <name> */

C Extra procedures for every class        /* class-method lookup table begins */
                                          void <name>_func_dispatcher() {
                                          /* Lookup Table */
                                          ...
                                          }
                                          /* class-method lookup table ends */
```

The broad translation for an object is described as follows:

```
IF1                                       nCube C
                                          /* Declaration */
X  0  <Class_name.Object_name>            SExp *<object_namename>;
C nodes(Simple or Compound), edges        /* Object <Class_name.Object_name>
                                             begins */
...
                                          void <Class_name_Object_name>() {
                                          ... }
```

The main.c is given as follows:

```
initialize all the objects;
initialize all threads or nodes;

switch(<Thread ID>) {
        case 1: <Object1_func>();
        break;
        case 2: <Object2_func>();
        break;
        case 3: <Object3_func>();
```

```
            break;
            case 4: <Object4_func>();
            break;
            case 5: <Object5_func>();
            break;
    }
    synchronize all threads to start at the same time;

    wait for terminations of all threads.
```

All the classes and objects are translated based upon the structures shown above, and correspondent portions of code are put into the header file or different c files to modularize the problem. The entire set of .c and .h files will be put into a directory according to what a programmer provides. Also a general makefile for the purpose of translation of target C code is given.

6.2.4 Additional Implementation Schemes

Unique data type and iteration conversion

The main feature of the PROOF language is to combine the functional and object-oriented domains together. In order to save the class and object information for the back-end translation, we also extend the IF1 to let it carry the class and object information across PROOF/L code to target C code. We apply the data structures, ATOM and SEXP, to realize all the functional features. Furthermore, lists used in the PROOF/L are a type of heterogeneous lists, which is similar to those in LISP. A number of list manipulation functions have been given, such as List Constructs [](square brackets), `append_left`, `head`, `tail`, `append_right`, `null?`, etc.

All other data types in the PROOF/L – integer, float, boolean, and string – are implemented with single type called ATOM with the unique type code embedded inside. The binary, boolean, relational and unary operations are applied in the following way:

```
verify type of the first atom depending on the operation;
extract content of the first atom;
verify type of the second atom depending on the operation if applicable;
extract content of the second atom if applicable;
apply the operation to content(s) of the atom(s);
compose an new atom with result of the operation and appropriate type;
```

The underling unique interface for processing different data types gives considerable flexibility to programmers, but sacrifices execution performance.

A functional language always involves recursion and PROOF/L is no exception. Recursive function calls not only are resource-consuming, but also limit computational capacity. Currently there is no implicit recursion removal during the translation. An iterative functional structure, WHILE loop, has been used to avoid these explicit recursive calls. Because of the specialty of WHILE loops, a library routine has been written to realize iterations of WHILE loops.

```
PROOF/L format:
        while(<predicate lambda exps>, <body lambda exps>) <an input to lambda>

Translated C format:
        result = <an input to lambda>;
        while (1) {
        if ( <predicate lambda exps> ( result ) ) {
                result = ( <body lambda exps> ( result );
        } else {
                break;
        }
        return result;
```

Commnumication and synchronization schemes

As mentioned in Section 6.1, distributed-memory parallel machines, like the nCube, need to explicitly specify communications among different physically separated nodes. On the other hand, shared-memory parallel machines, like the KSR, provide communications among different nodes at the kernel level, which releases this task from programmers. Our translator for nCube emulates communications for the shared-memory KSR machine in order to provide the unique structures to translate the IF1 code for two different target languages with as few variations as possible.

All the concurrent objects in PROOF/L that can be executed in parallel are dispatched to different threads or nodes. Each object is a computation unit of its own. The guard statements in methods are used to handle synchronizations among PROOF/L objects executed on different nodes or threads.

Periodic communications among different threads or nodes are made under these two conditions:

- call methods of other objects.
- applications of the parallel functions: `alpha`, `beta`, and `delta`.

The first condition is for the purpose of method invocations between two different objects. The reasons for one object to invoke another object's method are:

- Query for the state information of another object.
- Change the state of another object through the reception function (the only way to modify the state of an object)

The scheme for this kind of communication is shown in Figure 6.3. A request object sends the method id to indicate which method it wants to call to a responder object, assembles all the arguments necessary to that method, except state information of the responder object, into a stream packet, and then sends the packet through the network. After the method id received, the responder object dispatches the stream packet of arguments to the method that the request object requests. Arguments get extracted from the stream packet and passed to the method associated with state information of the responder object. The final outputs of the method are assembled again into a stream packet and passed back to the request object which in turn will disassemble the returned packet to get the results it expects. Overheads for assembling and disassembling are necessary for providing an unique and simple interaction between two different objects, and they are much less time-consuming than communication overheads across two different processors. Another reason for us to assemble all the arguments together and send once across two nodes is that multiple communications with small packets are more time-consuming than a single communication with a large packet.

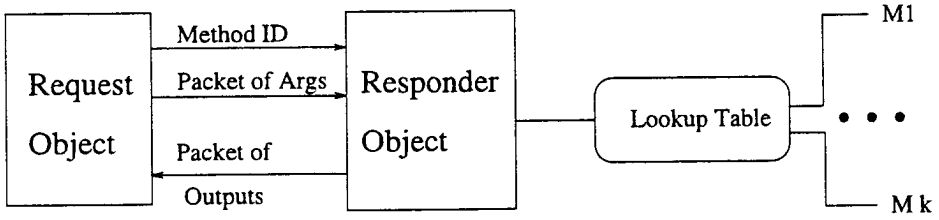


Figure 6.3: The underlying PROOF/L communication scheme

The code skeleton for a class-method lookup table is shown as the following:

```
/* message comes in */
switch (method_id) {
case 1:

method 1 set up;          /* receive argument packets and disassemble */
Call method 1;
method 1 feedback;        /* assemble final results */

break;

case 2:
...
}
```

Class-method lookup tables are required for the distributed-memory nCube machine. On the other hand, an object on the shared-memory KSR machine do not need to explicitly pass messages over to other objects; rather they can call another object's method directly because all objects shared their state information with others, while objects on the nCube machine own their state information themselves. But the sequence for calling methods is still the same, include assembling and disassembling arguments to methods.

Deadlock is entirely avoided. Each object controls its own thread by executing its body. The loose-coupled relations among objects are well maintained by all the GUARD statements within methods of each object. Objects are ready to serve other objects' requests when they enter unsatisfied GUARD statements.

Bottlenecks in the PROOF/L for a program normally are writable objects. When enormous objects want to invoke methods of writable objects, the program's execution pace slow down considerably.

The second condition is to execute a single function with multiple ranges of data in parallel(`alpha`, apply to all or `delta`, data partition) or multiple functions with their own data ranges in parallel(`beta`, distribute apply). The current translator can execute all the built-in functions and all the global functions in parallel.

Same reasons as the first condition are applied here for our assembling arguments to methods before calls and disassembling stream packets to extract results after methods finish executions.

- `alpha` function

```
PROOF/L format: alpha <method> ([args 1], [args 2], ...)
target C format:        assemble args 1;
                        launch a process to call method;
                        assemble args 2;
                        launch a process to call method;
                        ...
                        wait for return packet 1;
                        disassemble packet 1;
                        wait for return packet 2;
                        disassemble packet 2;
                        ...
                        build all return results into a list.
```

- `beta` function

```
PROOF/L format:         beta (method 1, method 2, ...) ([args 1],
                                                        [args 2], ...)
target C format:        assemble args 1;
```

```
launch a process to call method 1;
assemble args 2;
launch a process to call method 2;
...
wait for return packet 1;
disassemble packet 1;
wait for return packet 2;
disassemble packet 2;
...
build all return results into a list.
```

- delta function

PROOF/L format:

```
delta <method> ([low_bound], [upp_bound],
                <rest args> ...)
```

target C format:

```
assemble all args: low_bound, upp_bound,
                   rest args ...;
launch a process to call method;
launch a process to call method;
/* Depending on physical conpacity,
       launch processes as many as possible */
...
wait for return packet 1;
disassemble packet 1;
wait for return packet 2;
disassemble packet;
...
build all return results into a list.
    /* Dimension undermined */
```

Chapter 7

Extension of PROOF/L

In order to make PROOF/L more easy to use, we plan to add more functionalities to the original PROOF/L. In this chapter, we discuss how we are going to provide I/O functionality and an alternative data construct `array` in the PROOF/L for better utilizing the PROOF/L for developing parallel software in certain applications.

7.1 Input/Output in PROOF/L

As we mention in Chapter 1, PROOF/L is a C++ based language. In order to provide I/O functionality to the PROOF/L, we are going to build C++like system I/O classes to facility input/output for PROOF/L. The `read` and `write` operations should be sufficent for performing most I/O operations in PROOF/L. Additional functions for maintaining the files are needed, such as `open`, `close`, `rewind` and `seek`. We have defined the following four classes for any I/O processing:

- `fileIO` is an abstract data type which has the filename as a local variable, file manipulation operation, and I/O operations, such as `open`, `close`, `rewind`, `seek`, `read`, and `write`.
- `stdIO` is an abstract data type which has the standard I/O, STD, as a local variable for representing the standard I/O devices, and two standard I/O operations: `read` x and `write` x.
- `objectIO` is an abstract data type which has the *object-name* as a local variable, and the I/O operations, such as `read` x and `write` x, in which x is a variable to read from or write to. The reason for us to provide the `objectIO` class is to give users the capability to communicate among objects allocated over different processors. We try to give a unique interface to access objects, which is similar for access of standard I/O and files.

Each class has the following format:

```
class fileIO

   composition
      filename : string;
   end composition

      method  read  x;
      method  write x;

      method  open;
      method  close;
      method  rewind pos dir;
      method  seek pos;
end class

class stdIO

   composition
      STD : built-in;
   end composition

      method  read x;
      method  write x;
end class

class objectIO

   composition
      object_name : string;
   end composition

      method read  x;
      method write x;
end class
```

Figure 7.1 shows an example about how object A can access object B's data. For example, an object A wants to read x from an object B. Object A will send a message to object B to invoke `read` operation in object B. Then B will read x and send x to A. If A wants to write x into B, then A will send a message to invoke `write` operation in B. If `write` operation is invoked, then B will get x from A and write x to B.

As an alternative approach, we can design I/O features using a hierarchical structure of classes, which is similar to system I/O class hierarchy provided by C++. This

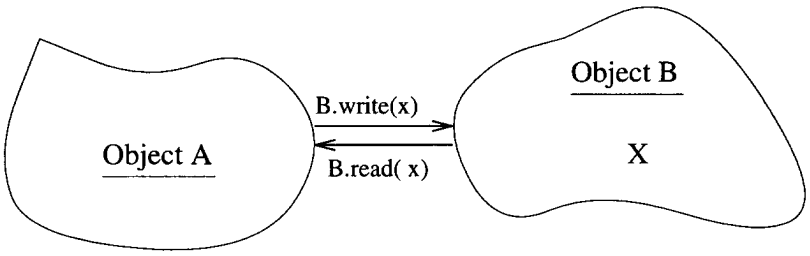


Figure 7.1: `Read` and `write` between two objects

structure consists of a set named class descriptions: `fileIO`, `stdIO`, and `objectIO`, which are organized as a superclass, such as `IO`. Using this structure, we can identify relations between classes and specify the inheritance, aggregation, and using relations among the classes. The hierarchy of the classes is shown in Figure 7.2:

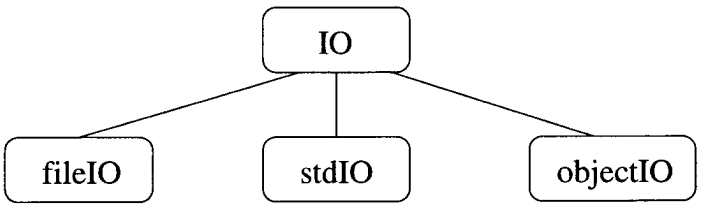


Figure 7.2: The hierarchical structure of classes `fileIO`, `stdIO`, `objectIO`, and `IO`

7.2 Arrays in PROOF/L

In this section, we will discuss how array data type can be incorporated in PROOF/L.

An array structure is a contiguous block of storage. Like a list, an array consists of elements of the same type. Unlike lists, an array is of a fixed, predetermined size which cannot be changed, but supports random access. PROOF/L needs to support the array data type to manipulate a group of elements to perform scientific computations.

7.2.1 Array Creation

The following syntax creates a one-dimensional array in PROOF/L.

```
array_name (array index element-type)
```

The *index* argument should be non-negative integers that are to be the number of elements of the array. The *element-type* is the name of the type of the elements of the array. It will be *integer*, *real*, and *boolean*. An array's index starts at 0.

A two-dimensional array can be created by the following syntax:

```
array_name (array index (array index element-type))
```

Multi-dimensional arrays are created by using nested structures and then *element-type* itself can also be an array.

7.2.2 Array Access

We have defined two functions: *retrieve* and *store*, which should be sufficient for accessing an array. Each element is referenced by its position. The function *retrieve* is normally used for accessing an element of an array. For example, we retrieve an element from a one-dimensional array as follows:

```
(retrieve array_name index)
```

The function *store* is used for filling an array's slot. This is a function of two arguments, where the first argument specifies a slot to be filled, and the second argument specifies the value to be stored in that slot. For example, the following statement stores an element into the one-dimensional array

```
(store array_name index element)
```

Let us consider the one-dimensional array

```
NAME_LIST array 100 int
- - - -
(store NAME_LIST 20 (retrieve NAME_LIST 5))
- - - -
```

In this example, *NAME_LIST* is an array of 100 integers. *retrieve* operation retrieves an element from an array *NAME_LIST* and *store* operation stores the element into the 21th slot.

For multi-dimensional arrays, we will use nested structures. For example, we can retrieve and store an element from a two-dimensional array as follows:

```
(retrieve (retrieve array_name index) index)
(store (store array_name index element) index)
```

Let us consider the two-dimensional array

```
MATRIX (array 100 (array 100 int))
- - - -
(store (store MATRIX 20
        (retrieve (retrieve MATRIX 10) 15)
                        ) 30)
- - - -
```

In this example, *MATRIX* is an array of dimensions 100 by 100. *ELEMENT* is an element in the 10th row and the 15th column of the array *MATRIX*. The element is stored into the *MATRIX* with the 20th row and the 30th column.

Chapter 8

Integration With Existing Languages

In order to utilize more mature FORTRAN/C built-in function libraries of parallel machines, we need to integrate PROOF/L with existing languages, such as FORTRAN and C.

8.1 C and FORTRAN on nCube and KSR Systems

The translation environment on KSR is shown in the Figure 8.1. *cc*, the KSR C compiler, translates a program written in C into executable load modules, or into a relocatable binary program that can subsequently be linked using *ld*. *cc* has the following syntax:

- On KSR: cc [option] sourcefile.c [-l lib]

option is a special action to be performed by *cc*. Multiple *option* parameters should be space-separated. For detailed information, see the *cc*(1) *man* page. The *sourcefile* is a source program to be compiler. Multiple *sourcefile* parameters should be space-separated. *cc* accepts several types of source files, and determines the action to take based on the filename's suffix:

- .c - a C source program to be compiled by the C compiler.
- .f/.F/.cmp - a FORTRAN source program to be compiled by the FORTRAN compiler (.cmp is related to FORTRAN's KSR KAP preprocessor).

The -l *lib* is an object library with which the program should be linked using *ld*. Multiple -l *lib* parameters should be space-separated. For more information on libraries, see Section 1.2, "Libraries"[67]. For example, the following *cc* command compiles a C program called *mainprog.c*. It specifies that the name of the executable file should be *mainprog* and that the program should be linked with the math library:

```
$ cc -o mainprog mainprog.c -lm
```

ncc, the nCube C compiler, has the following syntax:

On nCube: ncc [options] sourcefile.c [-l libraries]

For example, the text of the sample program used, *hello.c*, is as follows:

```
main() {
     printf ("Hello world!\n");
}
```

To compile the program *hello.c*, type the command line from a host shell:

```
$ ncc -d 2 hello.c
```

The **-d** option specifies that the program is to run on a 2-dimensional cube, that is, a set of four processors. To run the sample program, invoke it by typing the default name of the output file **a.out**. The output becomes:

```
$ a.out
Hello world!
Hello world!
Hello world!
Hello world!
```

The translation environment of both systems as shown in Figure 8.1 can call library functions at the linker stage. The KSR C compiler translates programs written in C into executable load modules. *Phread libraries*, *Unix libraries*, and *Presto libraries* are linked at the LINKER stage. The libraries are defined in /usr/lib/.

8.2 Integration of PROOF/L with nCube and KSR C/FORTRAN

The purpose of the integrating PROOF/L with existing languages is to use the resources of their existing software support. Any program, which is written in PROOF/L,

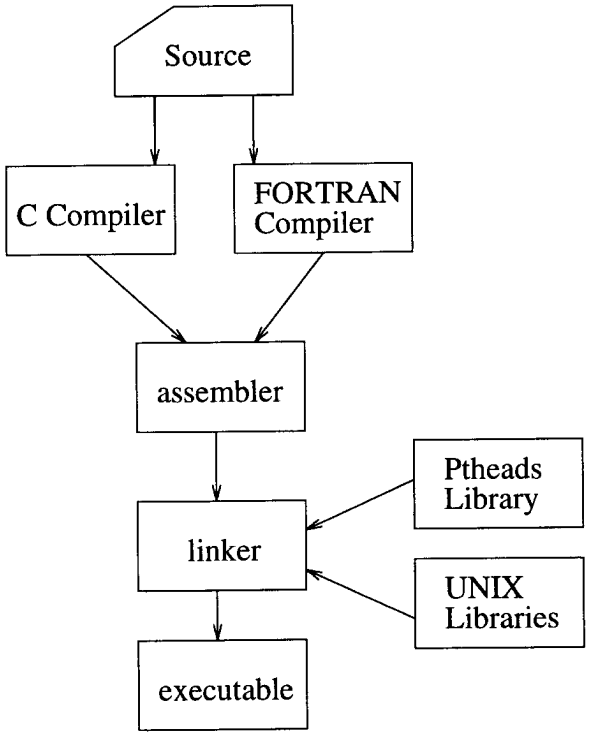


Figure 8.1: The KSR translation environment

can use the built-in functions written in FORTRAN and C. The type safe linkage scheme presents a problem if we try to call functions from other languages. We can specify the language type of a function, effectively turning off the name mangling. We create the following syntax in the PROOF/L program to specify function calls written in other languages:

```
extern language-type {
          #include <stdlib.h>;
          type func-name(arguments);
}
```

language-type is a language specification and *func-name* is a function name. For example, when you try to call a C function from PROOF/L, the linker will never find the function because the C function name is not mangled. You can declare a group of functions by using double quotes to enclose them in a linkage specification as follows:

```
extern 'C' { // here's a language specification
        #include <stdlib.h>;
        method func-name(args --> args);
}
```

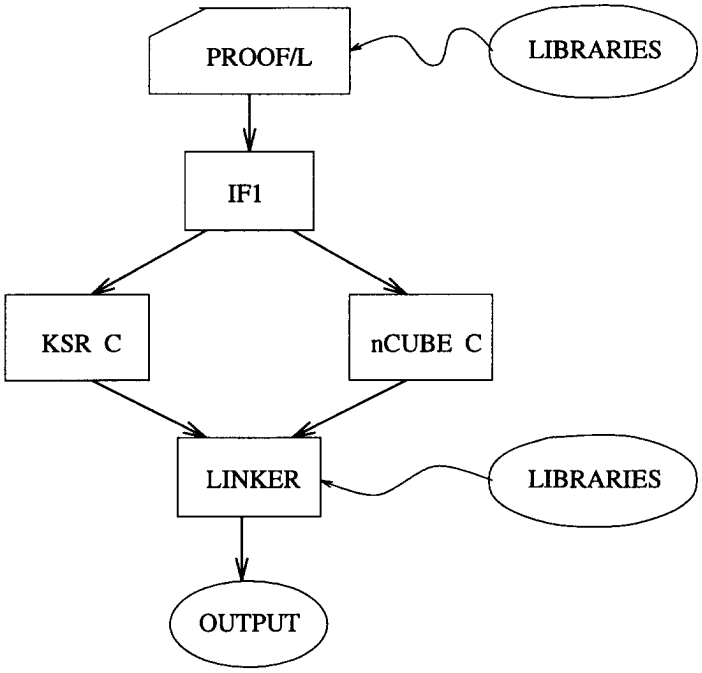


Figure 8.2: The translation environment for PROOF/L

The corresponding *IF1* with the above PROOF/L syntax will have the following forms:

```
I                40  func-name  %hf=stdlib.h
T  40  Function  41  42
C  input argument
T  41  - - - - -
       - - - - -
       - - - - - 0
C  output argument
T  42  - - - - -
       - - - - -
       - - - - - 0
```

func-name is a function name and *hf* means header files. The corresponding KSR C or nCube C program has the following format:

```
#include <stdlib.h>
    .
    .

  func-name(arguments);
    .
    .
```

The KSR C or nCube C program calls the function with arguments.

Figure 8.2 shows the translation environment for PROOF/L. Libraries are linked in PROOF/L at the LINKER stage. We created two translation rules, from PROOF/L to *IF1* and from *IF1* to nCube C or KSR C.

Chapter 9

Comparison of IF1 and IPR

In this chapter, we selected IF1 [5, 68] as the intermediate langauge for translating PROOF/L to any target language. We previously used the Intermediate Program Representation(IPR) [3], which was delevoped by ourselves, instead of IF1. The reason for us to switch from IPR to IF1 is to utilize the parallel software accessment tool – Parallel Accessment Window System (PAWS) [6] – for evaluating our approach since PAWS only accept IF1 code as input. In this chapter, we discuss the advantages and disadvantages of these two intermediate forms.

9.1 IF1

IF1 is one of several intermediate languages for functional language implementation, along with P-TAC [69] and Lean [70]. It is a data-flow intermediate form language based on acyclic graphs. IF1 was developed as an intermediate form for a high-level applicative programming language SISAL(Streams and Iteration in a Single-Assignment Language) [38]. IF1 is also a line-oriented language in that the line is the unit of construction of the IF1 files. IF1 consists of the following four components:

- Nodes: A node represents a logical or mathematical operation such as *addition, multiplication, subtraction, division, and, or* and *not.* There are two kinds of nodes:
  - Simple nodes: A simple node represents a basic operation such as addition. A node starts execution only after all the data carried by the incoming edges is available. Simple nodes do not contain subgraphs in them.
  - compound nodes: A compound node contains one or more subgraphs, and can be hierarchically defined.

  A node is created with the "N" code, followed by two integers: a node label and a node operation. Each node must be given a unique node identifier or label.

- Edges: An edge represents a data flow in the program. Edges connect nodes with each other. An edge is created with "E" code, followed by a five number tuple representing the source node, the source port, the destination node, the destination port and the literal type. A node may accept input from an edge which originates from another node or from a constant value or literal. A literal is created with the "L" code, followed by three numbers designating the destination port, the destination node and the type of the literal, followed by the actual literal value.
- Graph boundaries: A graph boundary is a border of the graph and it surrounds all the nodes and edges of the graph. Different graphs in an IF1 program are separated by graph boundaries. A graph boundary starts with the "G" code, followed by a type number.
- Types: Types are associated with edges. Every edge has a type, such as integer, boolean, array of integers, etc. There are six basic types in the IF1, and new types can be constructed from these basic types. A new type is created with the "T" code, followed by a unique identifying label, followed by a type entry code, followed by a basic type code. Records are formed in the IF1 by creating a record header (type entry 5) followed by a list of field entries (type entry 2).

9.2 IPR

IPR [3] language is designed to represent the parallelism in the PROOF/L program and analyze it for efficient exploitation on various parallel machines. The IPR consists of two different types of representation: one is a Petri-net and the other is a set of function nodes and their relations which we will introduce in this section. The semantics for the IPR is also given in two different levels: object level and method level. The object level semantics gives meaning to the object bodies of the PROOF/L program. The method level semantics gives meaning to the function nodes used to represent the methods in the PROOF/L program. This two level semantics makes it easy to understand the important issues in parallel programs, such as communication/synchronization aspects without considering the unnecessary details of the program. This separation of the semantics also allows the verification of programs in different levels, and thus the complexity of the understanding and the verification of programs can be significantly reduced.

IPR has a directed graph representation, in which the nodes represent computation, and edges represent data flow between nodes. Nodes can be divided into three types: computation nodes, control nodes and list handling nodes.

- Computation nodes: A computation node represents a function receiving input value(s) and generating output value(s). The computation nodes include basic *mathematical* and *boolean* operators, *constant, identity* and *copy* nodes. Mathematical and boolean operators includes operators such as $+, -, *, /, =, <, >$. The

constant node represents a constant generator. The identity node represents an identity function which always returns the same value as its input value. The copy node represents a duplicator, which produces copies having the same value as its input value.

- Control construct nodes: Control construct nodes are used to specify the control flow among functions. The control construct nodes include *select, distribute* and *merge.* The select node represents a conditional construction function. It receives input data $i_1, i_2, \ldots, i-n$ and control data c and returns an input i_i as an output according to the value of the control data c. The distributor node represents a conditional construction function. It receives input data i and control data c, and passes i to one of the output ports $o_1, o_2, \ldots, o_n$ according to the value of c. The merge node represents a nondeterministic selector, which receives an arbitrary number of input data sequentially and returns the one arriving first. If more than one input arrives at the same time, one of them is chosen arbitrarily.
- List handling nodes: There are two kinds of list handling nodes: *construct* and *split* nodes. The construct node receives one or more input values and make them as a list, and the split node receives a list as input and break down that list into values.

9.3 Comparison

In this section, we will compare IF1 and IPR in terms of data dependency representation, application areas, and maturity.

9.3.1 Data Dependency Representation

Both IF1 and IPR can explicitly represent the data dependencies among the nodes in the graphs, where the edges represent the data flow. Thus, they can easily represent explicit parallelism among the nodes. They can also be used to visualize the structure of the parallel programs. However, since these two intermediate languages are designed to be used for representing intermediate form of the program, it is not easy for compiler writer to read them. In case of IF1, because the names of the nodes are presented by integers, it is even more difficult for compiler writer to read IF1 than IPR.

9.3.2 Application Areas

IF1 is designed to be used as an intermediate representation for programs written in an applicative programming language SISAL[38]. IPR is designed to be used as an intermediate representation for programs written in a parallel object-oriented program-

ming language PROOF/L. Neither of the two intermediate forms is general enough to be used as a general intermediate representation language. However, considering that PROOF/L is an object-oriented programming language supporting the concepts such as object-orientation, persistent objects, it is more desirable to use an intermediate form which can represent such concepts supported in PROOF/L. IF1 does not support any of these concepts, but IPR supports such concepts. Thus, IF1 requires extensions to be used in the object-oriented program representation. Overall, since SISAL and IF1 were designed for numerical computation in scientific applications, its use for other application areas such as real-time systems and distributed systems would be difficult. On the other hand, IPR was designed to exploit the parallelism in PROOF/L programs for more general application areas, including scientific computation and simulation of concurrent or distributed systems. Because IPR can be used to represent synchronization and communication among the processes and shared data concept, it can also be used in the representation of the programs for real-time software systems and distributed systems. In addition, IPR can be used in the verification of the PROOF/L programs. Since the object-level relations among objects are represented in IPR by a Petri net, it is possible to verify the PROOF/L program in terms of synchronization and communication among the objects. Furthermore, the concurrency in the problem can be explicitly represented in IPR. IPR can also be used for the task allocation or scheduling analysis without augmentation.

9.3.3 Maturity

IF1 is more mature than IPR. IF1 has been studied and used as an intermediate form for the applicative programming language SISAL since 1985. There is a performance evaluation tool PAWS which only receives IF1 codes as input. On the other hand, IPR was used as an intermediate form for translation of PROOF/l to target codes. Both IF1 and IPR represent the program in the ASCII format. IF1 is more mature than IPR because IF1 has been fully implemented and used in the SISAL implementation.

Chapter 10

Conclusion and Future Research

In this project, we have completed the following tasks:

- Development of a front-end translator from PROOF/L to IF1
- Development of two back-end translators from IF1 to the C languages run on two different MIMD machines: nCube and KSR
- Evaluation of the effectiveness of our software development framework for parallel processing systems
- Comparison of IPR (Intermediate Program Representation) with IF1 (Intermediate Form 1)
- Extension of PROOF/L with input/output features and array construct
- Investigation on the integration of PROOF/L with existing programming languages, such as C and FORTRAN.

While existing approaches focus on developing software in the scientific computation area, our approach is suitable for general large-scale software development for parallel processing systems. Our approach is architecture-independent, and thus the programmers are free from explicitly specifying synchronization and communication. In addition, our approach is extensible for software development for distributed computing systems and/or real-time systems. We have also found that PROOF/L programs are generally shorter than their equivalent nCube C or KSR C programs. This indicates that the software development effort can be reduced when we develop software using our approach. We have used IF1 for the intermediate representation of PROOF/L. Since PROOF/L has some features that are not directly supported by IF1, we have added a number of constructs to IF1 to support PROOF/L's first-class functions, list-constructs and persistence. By building the two back-end translators, one for nCube which has a distributed memory architecture and one for KSR which has a shared memory architecture, we demonstrated that our approach can be implemented on

parallel machines with different architectures. In addition, the amount of implementation effort required to complete the second back-end translator turned out to be a small fraction of the effort required to complete the first back-end translator. Thus, the overhead to develop the back-end translators for various parallel machines can be kept small. Rather than translating PROOF/L to IF1 and then translating IF1 to the target code, we may have a single translator that translates PROOF/L to the target code directly. The single translation method may have the advantage that the front-end component and back-end component share common data structures, but has the disadvantage of reducing the architecture-independent portion of the translator.

In order to make our approach more practical, we need to improve our approach in the following aspects:

- Mapping: the partitioning and grain size determination approaches have been developed, but are not incorporated in the back-end translation. By fully implementing these approaches, we should significantly improve the overall performance of the PROOF/L programs.
- Translator: current implementation is a prototype, and thus requires additional optimizations to improve the efficiency of the generated target code. We need to implement more back-end translators for other parallel machines so that this approach can be used for various parallel machines.
- Extension to distributed computing systems: Currently, our approach is designed for software development for parallel processing systems, but can be extended for distributed computing systems.
- Development environment: By developing integrated CASE tools to support our approach, the software development effort can be further significantly reduced. Graphical user interface tool, display tool, and debugging tool are considered in this development environment.

Appendix A

PROOF/L Reference Manual

In this appendix, we present the PROOF/L reference manual which includes the syntax of PROOF/L and simple illustrative examples. Because PROOF/L is still under development, some features in this manual are our intention to complete in the near future, but not available in this release. Please refer to the last section of this appendix for information about the features that are supported in this release.

A.1 Introduction

PROOF/L (PaRallel Object-Oriented and Functional Language) is based on the PROOF computation model [4]. It is a C++-based language with additional constructs required in PROOF. In this appendix, we define the PROOF/L language, its syntax and semantics. At the end of this appendix is a section describing the current implementation of PROOF/L.

A.2 Structure of a PROOF/L Program

A PROOF/L program consists of a main program and a set of imported modules.

There are six parts in a PROOF/L program or module: preamble, import declarations, class declarations, object declarations, body declarations and initialization methods. We will discuss these parts here.

A.2.1 Preamble

The preamble of a PROOF/L program is

`program` *program_name* :

The preamble of a PROOF/L module is :

`module` *module_name* :

It is suggested that PROOF/L implementations require that the *program_name* and *module_name* identifiers match the name of the file name of the respective program and module files. Furthermore, it is suggested that PROOF/L programs end with the .PRF extension, and PROOF/L modules end with the .PMD extension.

A.2.2 Import Declarations

`import` *import-file_1*, *import-file_2*, ... *import-file_n*

Only the names of modules may appear in the import declaration. The compiler should report an error if the name of a program appears in the import declaration. No two modules may import each other. For instance, if module A imports module B, then module B may not import module A. All of the classes and objects of an imported module are available for use by the importing module. Mechanisms for data hiding will be defined in a subsequent report.

A.2.3 Class Declaration

In PROOF/L, every object is an instance of a class. A class is a template for a set of objects bearing similar behavior and is defined as a *generic abstract data type.*

A class in PROOF/L consists of a set of methods and a set of variable names representing the state of an object.

```
class Queue(itemtype)
   composition
      items : list(itemtype)
      numberOfItems : int
   end composition
```

```
    method Init(-> Queue)
       expression
          object (items = [],numberOfItems = 0)

    method AddItem(item : itemtype -> Queue)
       expression
          object (items = (append_right items item),
                  numberOfItems = (inc NumberOfItems))

    method GetTop(-> itemType)
       expression
           (head items)

    method RemoveTop(-> Queue)
       expression
          object Queue (items = (tail items),
                        numberOfItems = (dec NumberOfItems))

end class
```

Instance Variables

Sometimes it is useful to parameterize portions of the class specification. For instance, we may wish to define a class that serves as a queue for various kinds of objects (e.g.: integers, personnel records, banking transactions, restaurant orders). Rather than creating a separate class for each type of object, we can create a *parameterized* class using *instance variables.*

When an object is declared, the values of the instance variables must be defined. (see Section A.2.4).

The instance variables of an object may not be altered after object instantiation.

When a subclass is declared, the instance variables of the subclass must include all of the instance variables of its parent. The compiler should return an error if the instance variables of a superclass do not appear in the subclass declaration.

Composition

Since objects are persistent in PROOF/L, a data structure defining the composition of an object is declared in the *composition* section. The composition of an object consists of a collection of variables of various types. The variable can be a member of

```
class BoundedQueue(itemtype:type, maxitems : int) of Queue
   method AddItem(item : itemtype -> Queue)
      guard
         (< numberOfItems maxItems)
      expression
         object (items = (append_right items item),
                 numberOfItems = (inc NumberOfItems))
end class
```

the built-in types (See Section A.5) or an instance of a previously declared class.

Inheritance

Inheritence is used to define a subclass as a specialization of a superclass. In a subclass, the composition and methods of the superclass are inherited. Additional composition data and methods may be defined. Furthermore, inherited methods may be overriden by defining a new definition of the method.

A.2.4 Object Declaration

Once a class has been declared, objects may be instantiated in the object declaration section.

There are two types of objects in PROOF/L: active and passive. Active objects are associated with an object body that is declared in the body declaration section. The object body may be a non-terminating function. Passive objects are objects that are not associated with an object body. The methods of passive objects are called by the bodies of other objects.

```
passive object queue1 : instance of queue(integer)
passive object queue2 : instance of queue(float)
passive object queue3 : instance of queue(order)

active object sorter : instance of sorter
```

A.2.5 Body Declaration

Typically, an active object will wait for messages from other objects. The compiler should return a syntax error if an active object is instantiated without declaring a

body for the object.

```
body of server1 :
```

A.2.6 The Initialization Method

Before the methods for the bodies of active objects are evaluated, the program initialization method is invoked.

Although initialization of objects can occur in the bodies of the object, sometimes it is necessary to initialize objects in this section. For example, refer to the following code fragment:

```
body of object object1 :
   ;(R[|object1|](init),          % reception function # 1
     R[|object2|](do_something))  % reception function # 2

body of object object2 :
   R[|object2|](init)  % reception function # 3
```

Although one might expect that reception function #3 would be evaluated before reception function #2 is called, this in fact is not guaranteed by the PROOF/L language. If function #2 is called before function #3 has been evaluated, function #2 will attempted to modified an uninitialized object.

To avoid this problem, the objects should be initialized in the program initialization method:

```
begin
   ;( R[|object1|](init),
      R[|object2|](init))
end
```

A PROOF/L module may also have a program initialization method. The initialization method of a module will be executed before the initialization method of the program or module that imports the module.

Consider the program represented in Figure A.1. The main program imports modules 1, 2 and 3. Module 1 imports modules 4 and 5; module 2 imports modules 5 and 6; module 3 imports modules 6 and 7.

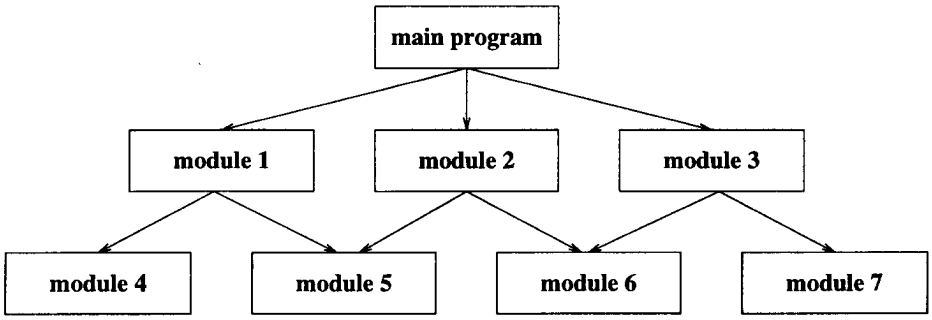


Figure A.1: A PROOF/L program with modules

When the program in executed, the initialization methods for modules 4-7 would be executed first; then the initialization methods for modules 1-3 would be executed. Finally, the program initialzation method would be executed.

There may be cases where a program or module does not require an initialization section.

An empty program or module initialization method is specified as follows:

```
begin
end
```

or

```
end
```

A.3 Method Declarations

Methods are declared within class declarations and global method blocks. There are two types of methods: pure applicative methods and modifier methods. Only applicative methods may occur within a global method block.

A method has the following syntax:

method name *(method-io) guard-dcln* **expression** *func*

A.3.1 Applicative Methods

Pure applicative methods are methods which do not change the state of an object. Applicative methods may neither invoke the *reception function*, nor invoke another method which invokes the *reception function.*

A.3.2 Modifier Methods

Modifier methods may invoke the *reception function*, applicative methods or other modifier methods. The reception function attaches data to an object as defined in the object's *composition.* Since modifier methods can change the state of an object, thus applicative semantics do not apply to modifier methods. However, by limiting the places where the reception function may be invoked, applicative semantics still apply for well-defined portions of a PROOF/L program.

A.3.3 Global Method Declaration

Sometimes the programmer may wish to define methods that are not associated with a particular class. For instance, it may be convenient to define routines such as `IsPrime`, `Double` and `Summation` that are available to several disjoint classes. [1]

A global method may be called by any other method.

A global method call can occur in another global method, a class method, the body of an object, or the program initialization method. Any global method calls must occur after the definition of the global method. Currently, forward declaration is not allowed in PROOF/L. Any number of methods may be declared in the **global/end global** block as follows:

```
global
   method double(number : int -> int)
      expression
         (+ number number)
end global
```

Global methods blocks may be interspersed with class declarations.

[1] Our inclusion of global methods in PROOF/L is not meant to subvert the object-oriented programming paradigm. By including global methods in our definition, we have in fact made it much easier to write non-object oriented programs; we can only hope that the PROOF/L programmer will recognize the numerous advantages of an OO approach. If everything in PROOF/L were represented as an object (for instance, in Smalltalk, even integers are objects), global methods would be less important.

A.3.4 Object Synchronization

Synchronization among objects is achieved by attaching an optional precondition, or *guard* expression, to class methods. The object which invokes the method is suspended when the attached guard expression evaluates to `False`, and resumed when the guard becomes `True`.

The guard expression must comply with the following rules:

1. It must be an applicative function that returns either `True` or `False`.
2. It may not invoke the *reception* function.
3. It may only refer to the data in the composition of the method's class; it may not refer to the data in the composition of other objects.
4. It may not invoke a class method. It may, however, invoke global methods.

A guard may only be attached to class methods and modifier methods. A guard may not be attached to a global method.

A guard is attached to a method by placing a **guard** keyword and accompanying applicative expression after the method's input-output declaration and before the method's **expression** clause.

Example:

```
class GumballMachine
   composition
      numberOfGumballs : int
   end composition

   method DispenseGumball( -> Gumball)
      guard
        (> numberOfGumballs 0)
      expression
        object Gumball(size = 1)

   method FillMachine( addGumballs : int -> GumballMachine )
      expression
        object GumballMachine(+ numberOfGumballs addGumballs)

end class
```

In the above example, when an object invokes the `DispenseGumball` method and there are no gumballs in the machine, the calling object will suspend execution until gumballs are added to the machine. To add gumballs to the machine, another object must modify the composition of the object with the reception function.

Consider instance `MyGumballMachine` of class `GumballMachine`:

```
R[|MyGumballMachine|](MyGumballMachine.FillMachine 100)
```

A.4 Comments

When a **#** character is placed on a line, the parser will ignore the **#** and the remaining characters after the **#** on the same line. The parser will also ignore blocks of text that appear within `/*` and `*/`.

A.5 Data Types

A.5.1 The `boolean` Type

The truth values are textually represented in a PROOF/L program as **true** and **false**.

A number of built-in functions and special forms return boolean values.

A.5.2 The `int` Type

An integer is a member of the set { ..., *-2, -1, 0, 1 , 2,* ... }.

Currently, the lower and upper bounds of integers are not defined by this document and are implementation dependent.

A.5.3 The `real` Type

Refer to the formal syntax rules of PROOF/L for detailed information on how reals are represented.

Examples:

```
1.34
1.10E-01
3.14159
```

Currently, the lower and upper bounds of real numbers are not defined by this document and are implementation dependent.

A.5.4 The `string` Type

Examples:

```
'this is a string'
'PROOF/L'
'1234'
```

Currently, the upper limit of the length of strings is not defined by this document and is implementation dependent.

A.5.5 The `list` Type

The list type of PROOF/L is similar to the list of functional languages such as LISP and Scheme.

A list in PROOF/L is defined recursively as follows: A list is either empty or non-empty. Non-empty lists are represented as a structure with two fields, *head* and *tail*. The *head* of a list may refer to any PROOF/L object (including another list), whereas the *tail* of a list must refer to another list.

PROOF/L provides a number of facilities for list creation and modification (see Section A.7.4). The *list-creation* function creates a new list. A null list is textually represented in PROOF/L as:

```
[ ]
```

Non-null lists are created using function.

```
[ 1 ]
[ 1 2 3 ]
[ 1 2 3 4 5 ]
```

The list-creation function may be nested.

```
[ [ 1 2 ] [ 1 2 ] [ 3 4 ] ]
[ [ [ 1 ] ] ]
[ 1 2 [ 3 4 ] [ 5 6 ] ]
```

Improper lists are lists which the tail of the list does not point to another list structure or **NULL**. *Improper lists* are not supported in PROOF/L.

A.5.6 User-defined Types

User-defined types are defined using the **class** construct (see below). When a class is defined, there exists an opportunity to define methods that work with instances of the class. Although some languages provide a separate mechanism to declare data types without associated methods (e.g., the **typedef** construct of C/C++), there is no such construct provided with PROOF/L.

A.6 PROOF/L Special Forms

The following section describes the PROOF/L special forms. *Special forms* are functions with syntactical forms that differ from the syntactical form used by methods.

Built-in functions (such as **inc** and **dec**) use the same syntax as methods; if they were not defined in PROOF/L, they could be defined as global methods.

A special form, however, has a syntax that differs from that of methods and subsequently cannot be declared by the user.

A.6.1 List Construction

A list can be constructed using the list construction operator.

Examples:

```
[ 1 2 3 ]
[ [ 1 2 ] [ 3 4 ] ]
[ [ 1 2 3 ] [ 4 5 6 ] ]
```

A.6.2 Alpha Function

The alpha function is syntactic sugar; the alpha expression:

`alpha` *function-name* [arg_1 arg_2 ... arg_x]

is equivalent to

[(*function-name* arg_1) (*function-name* arg_2) ...
(*function-name* arg_x)]

when the function has a single argument. When the function has n>1 arguments,

`beta[` *function-name*] [[$a_{1,1}$ $a_{1,2}$ ... $a_{1,n}$] [$a_{2,1}$ $a_{2,2}$ ... $a_{2,n}$]
... [$a_{x,1}$ $a_{x,2}$ ... $a_{x,n}$]

is equivalent to

[(*function-name* $a_{1,1}$ $a_{1,2}$ ... $a_{1,n}$)
(*function-name* $a_{2,1}$ $a_{2,2}$ ... $a_{2,n}$)
(*function-name* $a_{x,1}$ $a_{x,2}$ ... $a_{x,n}$)]

If the function to be applied to the argument list accepts only one argument, the argument-list should consist of a list of items of whatever type the function expects.

Example:

`alpha[inc][1 2 3 4]` ⇒ `[ 2 3 4 ]`

If the function to be applied to the argument accepts more than one argument, the argument-list should consist of a list of lists. Each list in the argument list should contain an element for each of the function's arguments.

Example:

```
alpha + [ [ 1 2 ] [ 3 4 ] [ 5 6 ] ] ⇒[ 3 7 11 ]
```

Although parallelism needs not be expressed explicitly in PROOF/L, the alpha function may be thought of as an explicit parallel structure.

A.6.3 Beta Function

The beta function accepts a list of functions and a list of arguments:

beta[$function_1$ $function_2$ ... $function_x$] [arg_1 arg_2 ... arg_x]

For each pair of functions and arguments j, if $function_j$ accepts a single element, then arg_j must be an item of the type that $function_j$ expects. If $function_j$ accepts n¿1 arguments, arg_j must be a list of arguments $[\ arg_{j,1}\ arg_{j,2}\ arg_{j,n}\ ]$ where for each z, $arg(j, z)$ is of the same type as parameter z of $function_j$.

The beta function is syntactic sugar; an beta expression of the form above is equivalent to

[($function_1$ $arg_{1,1}$ $arg_{1,2}$ ... arg_{1,n_1})
($function_2$ $arg_{2,1}$ $arg_{2,2}$ ... arg_{2,n_2}) ...
($function_x$ $arg_{x,1}$ $arg_{x,2}$ ... arg_{x,n_x})]

A.6.4 Let Function

let id = $func_1$ in $func_2$

is equivalent to

apply lambda(id)($func_2$) to $func_1$

For instance,

```
let x = (+ 3 3) in (+ x x)
```

is equivalent to

```
apply lambda(x)(+ x x) to (+ 3 3)
```

A.6.5 Sequence Function

;(($func_1$ ...), ($func_2$ ...), ... ($func_n$ ...))

Evaluates each function $func_1$, $func_2$, ..., $func_n$. The result of the sequence operator is the result returned by the final function in the function list. The results of the functions other than the final function are discarded. If any of the the function can cause side effects (namely, if the *reception* function is called), each function in the list is evaluated sequentially. If none of the functions cause side effects, the functions may be evaluated in parallel. The decision to evaluate the functions in parallel is made by the PROOF/L implementation. Whether the functions are evaluated in parallel is not explicitly stated in a PROOF/L program.

A.6.6 Lambda Function

`lambda (id)` *func*

The lambda function creates a function.

```
method test(a : int -> int)
   expression
      while( lambda(x)(< (head x) 10),
             lambda(x)([ (+ (head x) 1) (+ (head x) (tail x))]) )
      [1 0]
```

Implementation Note:

The current implementation of PROOF/L only uses lambda expressions within the **while** function. Furthermore, the implementation only supports single arguments to lamdba expressions.

A.6.7 Object Function

`object id (` *inst-list* `)`

The **object** special form creates a new composition of an object. It does not create a new process associated with that object, nor does it modify the contents of an existing object. Frequently, the result of an **object** special form is used by the reception function to modify an object.

A.6.8 Apply Function

Apply is used to apply a *function* to its arguments. If the function accepts a single argument, the apply argument should be an item of the same type that the function expects. If the function accepts more than one argument, the apply argument should consist of a list of length n, where n is the number of arguments the function accepts, and each element in the list is of the same type of the corresponding argument of the function.

Examples:

```
apply inc to 1

apply lambda(x y)(+ x y) to [1 2]
```

A.6.9 Loop

`loop (` *func* `)`

The **loop** special form repeatedly evaluates *func*. The evaluation of the **loop** special form never terminates.

Example:

```
loop ( (out ''Hello world''))
```

The loop function is syntactic sugar; it is equivalent to

`while (` `True, lambda(`z`)`*func* `) []`

where z is an identifier not in *func*.

A.6.10 While Function

`while (` $func_1$ `,` $func_2$ `)` $func_3$

$func_1$ and $func_2$ must be lambda functions. $func_3$ can be any valid expression.

The **while** special form is defined recursively as follows:

```
while ( func1 , func2 ) func3 ⇒
if (apply func1 to func3,
while(func1,func2) apply func2 to func3,
func3)
```

A.6.11 If Function

```
if (func1 , func2 , func3 )
```

The **if** special form accepts three arguments. If $func_1$ evaluates to be true, then **if** returns the result of the evaluation of $func_2$; otherwise, **if** returns the result of the evaluation of $func_3$.

Examples:

```
if (> a b, a, b) ⇒(the maximum of a, b)

if ((> a b),
    if ((> a c),a,c),
    if ((> b c),b,c))

⇒(the maximum of a, b c)
```

A.6.12 Reception Psuedo-Function

The *Reception Psuedo-Function* is used to modify the composition of an object. Unlike other functions in PROOF/L, the *Reception Psuedo-Function* is not an applicative function since it alters the state of an object.

The Reception Psuedo-Function may only appear:

- within the body of an object
- within a modifier method
- within the program initialaztion method

The compiler should generate an error if the reception function appears anywhere else.

Table 1: A multi-mode locking mechanism.

	R-Lock	W-Lock	M-Lock
R-Lock	compatible	compatible	incompatible
W-Lock	compatible	incompatible	incompatible
M-Lock	incompatible	incompatible	incompatible

The *Reception Psuedo-Function* has the following form:

```
R[| name |] func
```

where **name** is the name of the object to be modified (the *recipient*) and **func** is a valid applicative PROOF/L function that returns a value of the recipient's class. The **func** must not include a reference to a *Reception Psuedo-Function* or a modifier method.

Example:

```
R[|object1|](object1.Init)
```

The major difference between modification of objects with the Reception Pseudo-Function and the traditional assignment statement are

- The evaluatation of the expression **func** can be in parallel since **func** contains only applications of purely applicative functions.
- PROOF/L provides a mechanism that prevents the simultaneous modification of objects.

PROOF/L prevents the simultaneous modification of objects with a three-mode locking mechanism. At any moment, an object involved in an expression is in one of the following three categories:

- **read-only** : The expression only needs to read the value of the object.
- **will-modifiy** : The expression will modify the object, but the modification does not occur at this moment.
- **modifying** : The expression is currently modifying the object.

The three types of locks, **R-Lock**, **W-Lock**, and **M-Lock** are associated with the three statuses of an object, **read-only**, **will-modify** and **modifying**, respectively. A lock

is granted only when it is compatible with other locks granted for the same object, according to the compatibility chart in Table 1.

Before the evaluation of `func` in the Reception Pseudo-Function, a `W-Lock` must placed on the recipient, if possible. If a `W-Lock` cannot be placed on the recipient immediately, the process waits until the lock can be placed. Once `func` is evaluated, a `M-Lock` is placed on the object and the object is modified.

A.7 Built-In PROOF/L functions and identifiers

A.7.1 Integer Manipulation Functions

+,-,*,\,mod

inc

dec

A.7.2 Real Manipulation Functions

+,-,*,\,mod

A.7.3 Numeric Conversion Functions

Function Name	Inputs	Outputs	Description
`floor`	`real`	`int`	floor
`ceiling`	`real`	`int`	ceiling
`trunc`	`real`	`int`	trunc
`round`	`real`	`int`	round

A.7.4 List Manipulation Functions

The following functions work with the **list** data-type, defined in section A.5.5.

head

The head function accepts a single argument, a list, and returns the head of the list.

Examples:

```
(head [1 2 3] ) ⇒1
(head [ [ 1 2 ] [ 3 4] ] ) ⇒[ 1 2 ]
(head (head [ [ 1 2 ] [ 3 4 ] ] )) ⇒1
```

tail

The tail function accepts a single argument, a list, and returns the tail of the list.

Examples:

```
(tail [1]) ⇒[]
(tail [1 2 3] ) ⇒[ 2 3 ]
(tail [ [ 1 2 ] [ 3 4] ] ) ⇒[ [ 3 4 ] ]
(tail (tail [ [ 1 2 ] [ 3 4 ] ] )) ⇒[]
```

last

The last function accepts a single argument, a list, and returns the last element in the list.

Examples:

```
(last [1 2 3] ) ⇒3
(last [ [ 1 2 ] [ 3 4 ] ] ) ⇒[ 3 4 ]
(last (last [ [ 1 2 ] [ 3 4] ] ) ) ⇒4
```

append_right

The *append_right* function accepts two parameters

(append_right arg_1 arg_2)

where arg_1 is a list of size n and arg_2 is a member of any type. It returns a list with $n+1$ parameters, where elements 1-n correspond to elements $1-n$ of arg_1 and with arg_2 as element $n+1$.

Examples:

```
(append_right [ 1 2 ] 3 ) ⇒[ 1 2 3 ]
(append_right [ ] 1 ) ⇒[ 1 ]
(append_right [ [ 1 2 ] ] [ 3 4 ] ) ⇒[ [ 1 2 ] [ 3 4 ]]
```

The *append_right* function requires that a deep-copy be made of the list passed in arg_1; use of *append_right* can be very costly and should be avoided when possible.

append_left

The *append_left* function accepts two parameters

(append_left arg_1 arg_2)

where arg_1 is a member of any type and arg_2 is a list. It returns a list with arg_1 as the first element followed by each of the elements of arg_2.

Examples:

```
(append_left 4 [1 2 3]) ⇒[4 1 2 3]
(append_left [1 2 3] [4 5 6]) ⇒[ [ 1 2 3] 4 5 6 ]
```

listref

(listref *list i*)

The *listref* function returns the i^{th} element of *list*. The first element of the list is numbered 0.

Examples:

```
(listref [1 2 3] 0) ⇒1
(listref [[1 2] [3 4]] 1) ⇒[3 4]
(listref (listref [[1 2] [3 4]] 1) 1) ⇒3
```

Since the list is a linked structure, the worst case access time for a list of n elements is O(n). An array, although not as flexible as a list, provides O(1) access time.

A.7.5 self

The **self** identifier is used with the methods of a class to access the composition of the class. Use of **self** is infrequent; it is usually used to return the entire composition of the class from a function.

Example:

```
class sample
   composition
      a : int
      b : int
   end composition
   method ChangeIfOne(number : int -> sample)
      expression
         if( (= number 1),
             object sample ( a = a + 10, b = b + 20 ) ,
             self )
   end method
end class
```

self may also be used to access a member in the composition when a variable name in the method declaration is the same as a variable name in the composition.

Example:

```
class sample
   composition
      a : int
      b : int
   end composition
   method ChangeIfOne(a : int -> int)
```

```
      expression
         (+ a self.a)
   end method
end class
```

A.7.6 super

The **super** identifier is used within the methods of a subclass to access the superclass. Usually it is used to access the overriden methods of superclass.

Example:

```
class superclass
   method DoSomething(->int)
      expression (+ 2 2)
end class

class subclass of superclass
   method DoSomething(->int)
      expression (+ 2 (super.DoSomething))
end class
```

In the above example, a call to *DoSomething* of *subclass* would result in the sum of 2 plus the result of calling the *DoSomething* method of superclass.

A.8 Implementation Notes

The current implementation of PROOF/L does not support a number of features described in this chapter:

- Inheritance
- Modules
- Modifier methods
- Arrays
- Input-Output Functions
- Lambda functions outside of `while` functions.

Appendix B

Syntax of PROOF/L

The following is a formal syntax for PROOF/L in extended BNF. This is an expanded version of the syntax presented in [3]. Terminals appear in **bold**. Non-terminals appear in *italics*.

main-program → **program** *name* : *import-list class-list obj-list body-list program-body*

module → **module** *name* : *import-list class-list obj-list body-list program-body*

import-list → *more-imports name*

more-imports → *name* , *more-imports*
|

class-list → *class-def class-list*
|

class-def → **class** *name class-ins super-class compostion method-def* **end class**
| **global** *method-def* **end global**

class-ins → *(dcln-list)*
|

program-body → **begin** *func* **end**
| **begin end**
| **end**

body-list → *body-def body-list*
|

body-def → **body of object** *name* : *func*

obj-list → *obj-list obj-def*
| |

obj-def → *active-opt* **object** *name-list* : **instance of** *name ins-opt*

ins-opt → *(name-list)*
| |

active-opt → **active**
| **pseduo**
| **passive**
| |

super-class → **of** *name*

name-list → *name* , *name-list*
| *name*

composition → **composition** *var-list* **end composition**
| |

var-list → *var-list dcln*
| |

method-def → *method method-def*
| |

method → **method name** *(method-io)* *guard-dcln* **expression** *func*
| **external** *string* **method-def** *(method-io)* { *include-list* }

guard-dcln → **guard** (*bool-exp*)
| |

method-io → *input-list -> output-list*

input-list → *dcln-list*

output-list → *dcln-list*

dcln-list → *the-dcln-list*
| |

the-dcln-list → *dcln* , *the-dcln-list*
| *dcln*

dcln → *name* : *data-type*
| *name* : *class-name*
| *name* : *list-opt* (*data-type*)

| *name* : *list-opt* (**name**)
| *name*
| *data-type*
| **list**
| *list-opt* (*data-type*)

data-type → **int**
| **boolean**
| **real**
| **array** [**integer**] **of** *data-type*

list-opt → **list** ∗ *list-opt*
| **list**

class-name → *name*

inst-list → *inst* , *inst-list*
| *inst*

inst → *name* = *func*

func → **alpha** *name* [*func-list*]
| **beta** [*name-list*][*func-list*]
| **delta** (*name func-list*)
| **let** *name* = *func* **in** *func*
| ;(*func-list*)
| **lambda** (*name*) *func*
| **object** *name* (*inst-list*)
| **apply** *func* **to** *func*
| **loop** (*func*)
| **while** (*func, func*) *func*
| **if** (*func, func, func*)
| **R**[| *name* |] *func*
| (*func-list*) *(Function Call)*
| [*func-list*] *(List construction)*
| *binop func func*
| *boolop func func*
| **not** *func*
| *prefix name*
| **integer**
| **float**
| **NIL**
| **string**
| **true**
| **false**
| **self**

prefix → *self*
| *super*
| *name*
|

func-list → *func-list* , *func*
|

binop → + *(Binary Operators)*
| −
| *
| /
| *mod*

boolop → = *(Boolean Operators)*
| !=
| <
| <=
| >
| >=
| **or**
| **and**

letter → **a** | **b** | **c** | ... | **z**

digit → **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **0**

underscore → _

name → *letter* (*letter* | *digit* | *underscore*)*

sign → + | -

float → (*number*)+ . (*number*)*
| (*number*)+ . (*number*)* **E** *sign* (*digit*)+

string → ' (*string-element*)* '

string-element → <*any character other than* ' >

include-list → (*include-char*)*

include-char → <*any character other than* } >

Bibliography

[1] S. S. Yau, X. Jia, D-H. Bae, M. Chidambaram, and G. Oh, "An Object-Oriented Approach to Software Development for Parallel Processing Systems," *Proc. 15th Int'l Computer Software & Applications Conf. (COMPSAC 91)*, September 1991, pp. 453-458.

[2] S. S. Yau, D.-H. Bae and M. Chidambaram, "A Framework for Software Development for Distributed Parallel Computing Systems, " *Proc. Third Workshop on Future Trends of Distributed Computing Systems,* April 1992, pp. 240–246.

[3] S. S. Yau, D.-H. Bae, M. Chidambaram, G. Pour, V. R. Satish, W-K. Sung and K. Yeom, "Software Engineering For Effective Utilization of Parallel Processing Computing Systems," RL-TR-93-113, Final Technical Report, Rome Laboratory, Air Force Material Command, Griffiss Air Force base, New York, June 1993.

[4] S. S. Yau, X. Jia, and D.-H. Bae. "PROOF: A Parallel Object-Oriented Functional Computation Model," *Journal of Parallel and Distributed Computing*, Vol. 12, No. 3, July 1991, pp. 202–212

[5] Livermore National Laboratory, *An Intermediate Form Language IF1,* Reference Manual 1985.

[6] D. J. Pease, "Parallel Computing Systems", Final Technical Report, Rome Laboratory, Air Force Material Command, Griffiss Air Force base, New York, June 1992.

[7] J. C. Browne, M. Azam and S. Sobek, "CODE: A Unified Approach to Parallel Programming," *IEEE Software*, Vol. 6. No. 4, July 1989, pp. 10–18.

[8] K. Zink and J. C. Browne, "Design Approach for High Performance Computing," PD-101, Final Technical Report, Rome Laboratory, Air Force Material Command, Griffiss Air Force base, New York, March 1993.

[9] P. H. Mills, L. S. Nyland, J. F. Prins, and J. H. Reif, "Prototyping N-body Simulation in Proteus," *Proc. Sixth Int'l Parallel Processing Symposium,* March 1992, pp. 476–482.

[10] R. Jagannathan, A. R. Downing, W. T. Zaumen, T., and R. K. S. Lee, "Dataflow-based Methodology for Coarse-grain Multiprocessing on a Network of Workstations," *Proc. Int'l Conf. on Parallel Processing*, Vol. II, 1989, pp. 209-219.

[11] J.-L. Gaudiot and L.-T. Lee, "Occamflow: A Methodology for Programming Multiprocessor Systems," *Journal of Parallel and Distributed Computing*, Vol. 7, 1989, pp. 96–124.

[12] F. Baiardi, et al, "Pisa Parallel Processing Project on General-Purpose Highly-Parallel Computers," *Proc 15th Annual Int'l Computer Software & Applications Conf. (COMPSAC'91)*, 1991, pp. 536–542.

[13] M. J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Trans. on Computers*, C-21, No. 9, September 1972, pp. 948-960.

[14] W. D. Hillis, *The Connection Machine,* MIT Press, Cambridge, Massachusetts, 1985.

[15] H. T. Kung. "Notes on VLSI Computation," in D. J. Evans, ed., *Parallel Processing Systems.* Cambridge University Press, 1982, pp. 40–62.

[16] K. Hwang, *Advance Computer Architectures,* McGraw Hill, 1993.

[17] K. Hwang. "Advanced Parallel Processing with Supercomputer Architecture," *Proc. IEEE*, Vol. 75, No. 10, 1987, pp. 1348–1379.

[18] J. Test, M. Myszewski, and R. C. Swift. "The Alliant FX/Series: Automatic Parallelism in a Multiprocessor Mini-Supercomputer," in W. J. Karplus, ed., *Multiprocessors and Array processors*, Simulation Councils Inc., 1987, pp. 35–44.

[19] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, K. P. McAuliffe, and E. A. Melton. "The IBM Research Parallel Processor Prototype (rp3)," *Proc. Int'l Conf. on Parallel Processing*, 1985, pp. 764–711.

[20] S. Thakkar, P. Gifford, and G. Fielland. "Balance: A Shared Memory Multiprocessor," *Proc. Second Int'l Conf. on Supercomputing*, 1987, pp. 23–30.

[21] J. Graham and J. Rattner. "Expert Computation on the iPSC Concurrent Computer," in W. J. Karplus, ed., *Multiprocessors and Array processors*, Simulation Councils Inc., 1987, pp. 167–176.

[22] J. F. Palmer. "The NCUBE Family of Parallel Supercomputers," in W. J. Karplus, ed., *Multiprocessors and Array processors*, Simulation Councils Inc., 1987.

[23] C. Y. Chin and K. Hwang. "Packet Switching Networks for Multiprocessors and Dataflow Computers," *IEEE Trans. on Computer*, Vol. C-33, No. 9, 1984, pp. 991–1003.

[24] Inmos Ltd., *The Transputer Databook,* Inmos Ltd., Bristol, UK, 1989.

[25] Cray Research Inc., *Fortran(CFT) Reference Manual*, 1984.

[26] Ardent Computer Co., *Programmer's Guide*, 1989.

[27] C. Huson, T. Mache, J. Davies, M. Wolfe and B. Leasure, "The KAP-205: An Advanced Source-to-Source Vectorizer for the Cyber 205 Supercomputer," *Proc. Int'l Conf. on Parallel Processing*, 1986, pp. 827–832.

[28] P. R. Fenner, "The Flex/32 for Real-Time Multicomputer Simulation," in W. J. Karplus, ed., *Multiprocessors and Array Processors*, Simulation Councils Inc., 1987, pp. 127–136.

[29] F. Allen, M. Burke, P. Charles, R. Cytron and J. Ferrante, " An Overview of the PTRAN Analysis System for Multiprocessing," *Journal of Parallel and Distributed Computing,* Vol. 5, No. 5, 1988, pp. 617–640.

[30] Z. Bozkus, *et al*, "Compiling Distribution Directives in a Fortran 90D Compiler," *Technical Report SCCS-388*, Northeastern Parallel Architecture Center, July 1992.

[31] C. M. Chase, A. L. Cheung, A. P. Reeves and M. R. Smith, "Paragon: A Parallel Programming Environment for Scientific Applications Using Communication Structures," *Journal of Parallel and Distributed Computing*, Vol. 16, 1992, pp. 79–91.

[32] C.A.R. Hoare, "Communicating Sequential Processes," *Comm. ACM*, Vol. 21, No. 8, August 1978, pp. 666-677.

[33] Department of Defense, *Reference Manual for the Ada programming Language*, ANSI/MIL-STD-1815A-1983, 1983.

[34] N. Carriero and D. Gelernter, "Linda in Context," *Comm. ACM*, Vol. 32, No. 4, 1989, pp. 444-458.

[35] A. S. Grimshaw, "Easy-to-Use Object-Oriented Parallel Processing with Mentat," *Computer*, Vol 26, No. 5, 1993, pp. 39–51.

[36] D. Gannon and J. K. Lee, "Object-Oriented Parallel Programming Experiments and Results," *Proc. 1991 Supercomputing Conf.*, 1991.

[37] F. B. Irisa, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony and B. Mohr, "Implementing a Parallel C++ Runtime System for Scalable Parallel System," *Proc. 1993 Supercomputing Conf.*, 1993.

[38] J. McGraw, *et al*, *SISAL: Streams and Iteration in a Single Assignment Language*, Language Reference Manual Version 1.2, 1985.

[39] K. L Clark and S. Gregory, "PARLOG: Parallel Programming Logic," *ACM Trans. on Programming Languages and Systems*, Vol. 8, No. 1, 1986, pp. 1–49.

[40] H. Liberman, "Concurrent Object-Oriented Programming in Act 1," in Yonezawa and M. Tokoro(eds.), *Object-Oriented Concurrent Programming*, MIT Press, 1987, pp. 9–36.

[41] S. S. Yau, X. Jia, and D.-H. Bae, "Trends in Software Design for Distributed Computing Systems," *Proc. Second Workshop on the Future Trends of Distributed Computing Systems*, October 1990, pp. 154–160.

[42] R. W. Sebesta, *Concepts of Programming Lanuages,* Benjamin/Cummings Publishing Company, 1992.

[43] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

[44] P. Coad and E. Yourdon, *Object-Oriented Analysis*, Yourdon Press, 1991.

[45] K. S. Rubin and A. Goldberg, "Object Behavior Analysis," *Comm. ACM,* September 1992, Vol. 35, No. 9, pp. 48–62.

[46] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[47] D. E. Eager J. Zahorjan and E. D. Lazowska, "Speedup Versus Efficiency in Parallel Systems," *IEEE Trans. on Computers*, Vol. 38, No. 3, 1989, pp. 408–423.

[48] H. S. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Trans. on Software Engineering*, Vol. SE-3, No. 1, 1977, pp. 85–93.

[49] C. C. Shen and W. T. Tsai, " A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion," *IEEE Trans. on Computers*, Vol. 34, No. 3, 1985, pp. 197–203.

[50] W. W. Chu, L. J. Holloway, M.-T. Lan, and K. Efe, "Task Allocation in Distributed Data Processing," *IEEE Computer*, Vol. 13, No. 11, 1980, pp. 57–69.

[51] O. I. El-Dessouki and W. H. Huan, "Distributed Enumeration on Network Computers," *IEEE Trans. on Computers*, Vol. C-29, No. 9, 1980, pp. 818–825.

[52] K. Efe, "Heuristic Models of Task Assignment Scheduling in Distributed Systems," *IEEE Computer*, Vol. 15, No. 6, 1982, pp. 50–562.

[53] P. R. Ma and E. Y. S. Lee, "A Task Allocation Model for Distributed Computing Systems," *IEEE Trans. on Computers*, Vol. C-31, No. 1, 1982, pp. 41–472.

[54] V. M. Lo, "Task Assignment to Minimize Completion Time," *Proc. IEEE 5th Int'l Conf. on Distributed Operating Systems*, 1985, pp. 329–336.

[55] S. M. Shatz, and S. S. Yau, " A Partitioning Algorithm for Distributed Software Systems Design," *Information Sciences*, Vol. 38, No. 2, 1986, pp. 165–180.

[56] S. S. Yau and I. Wiharja, "An Approach to Module Distribution for the Design of Embedded Distributed Software Systems," *Information Sciences*, Vol. 56, 1991, pp. 1–22.

[57] S. S. Yau, D.-H. Bae, and Gilda Pour, "A Partitioning Approach for Object-Oriented Software Development for Parallel Processing Systems," *Proc. 16th Annual Int'l Computer Software & Applications Conf. (COMPSAC92)*, October 1992, pp. 251–256.

[58] S. S. Yau and V. R. Satish, "A Task Allocation Algorithm for Distributed Computing Systems," *Proc. 17th Annual Int'l Computer Software & Applications Conf. (COMPSAC93)*, November 1992, pp. 336–342.

[59] C. N. Nikolaou and A. Ghafoor, "On the Assignment Problem of Arbitrary Process Systems to Heterogeneous Distributed Computer Systems," *IEEE Trans. on Computers*, Vol. 41, No. 3, March 1992, pp. 257–273.

[60] D. Fernandez-Baca, "Allocating Modules to Processors in a Distributed Systems," *IEEE Trans. on Software Engineering*, Vol. 15, No. 11, November 1989, pp. 1427–1436

[61] S. M. Shatz, J-P Wang and M. Goto, "Task Allocation for Maximizing Reliability of Distributed Computer Systems," *IEEE Trans. on Computers*, Vol. 41, No. 9, September 1992, pp. 1156-1168.

[62] G. Goldberg, "Multiprocessor Execution of Functional Program," *Journal of Parallel Programming*, Vol. 17, No. 5, 1988, pp. 425–473.

[63] I. Foster, *New Concepts in Parallel Programming,* Prentice Hall, Englewood Cliffs, NJ, 1990.

[64] S. S. Yau and G. H. Oh, "An Object-Oriented Approach to Software Development for Autonomous Decentralized Systems," *Proc. Int'l Symposium on Autonomous Decentralized Systems (ISADS 93)*, 1993, pp. 37–43.

[65] S. S. Yau, K. Yeom, Bing Gao, Ling Li and D-H. Bae, "An Object-Oriented Software Development Framework for Autonomous Decentralized Systems," *Proc. Second Int'l Symposium on Autonomous Decentralized Systems (ISADS 95)*, 1995, pp. 405–411.

[66] nCUBE Inc., *nCUBE 2 Programmer's Manual*, 1992.

[67] Kendall Square Research, Inc., *KSR C Programming*, 1993.

[68] S.K.Skedzielewski and M. L. Welcome, "Data Flow Graph Optimization in IF1," *Proc. Functional Programming Languages and Computer Architecture Conf.*, 1985, pp. 17-34.

[69] Z. Ariola and Arvind, "P-TAC: A Parallel Intermediate Language," *Proc. Functional Programming Languages and Computer Architecture Conf.*, 1989, pp. 230-242.

[70] H.P. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, M. J. Plasmeijer, and M. R. Sleep, "Towards an Intermediate Language based on Graph Rewriting,", *Proc. PARLE Conf., LNCS 259*, 1987, pp. 159-175.

*U.S. GOVERNMENT PRINTING OFFICE: 1995-710-126-20064

MISSION

OF

ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

a. Conducts vigorous research, development and test programs in all applicable technologies;

b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;

c. Provides a full range of technical support to Air Force Materiel Command product centers and other Air Force organizations;

d. Promotes transfer of technology to the private sector;

e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.