A Framework for Software Development for Distributed Parallel Computing Systems*

Stephen S. Yau, Doo-Hwan Bae and Madhan Chidambaram
Computer and Information Sciences Department
University of Florida
Gainesville, Florida 32611-2024, USA

Abstract

In this paper, a framework for software development for distributed parallel computing systems based on the parallel object-oriented functional computational model PROOF is presented. Our approach will enable programming to be independent of the configuration of the computing system on which the program will be executed. In our approach, the programmer does not need to be bothered by the parallelism in the application or the architectural details. These considerations can be handled at the translation and the allocation stages. Our approach retains the benefits of both the object-oriented as well the functional paradigms.

1 Introduction

During the last decade, rapid progress has been made in microelectronic technology, and the applications of the parallel and distributed computing systems such as process control and telecommunications systems are becoming more feasible. The trend is toward a distributed parallel computing system in which one or more of the sites are made up of parallel processing computing systems. This becomes feasible with the availability of low cost parallel processing systems, such as the transputers and other parallel processing machines. However, the development of the software for such systems has become rather complicated due to the architectural considerations of the distributed system, including the configuration of the parallel computing systems at individual sites in the distributed system. Interprocess and intraprocess communications, partitioning and allocation of the software system are all very important issues in such software development. Furthermore, because most of the existing parallel languages are tied to a specific architecture, the software development approach for each architecture becomes architecture dependent [1].

In this paper, we will present a framework for software development for distributed parallel computing systems based on the parallel object-oriented functional computational model PROOF [2]. Our approach is architecture transparent, i.e., the configuration details of the computing system are considered at the translation phase. The programmer is liberated from the issues such as parallelism and configuration of the computing system during software development. Our approach focuses on the performance of the software system by identifying and modifying the bottleneck objects and shared writable objects which limit the parallelism in the software system, if possible.

*The work was supported by the Rome Laboratory, AFSC under contract No. F30602-91-C-0045.

2 Our Approach

The computation model PROOF [2] was intended for software development for parallel processing system [3], and it incorporates the functional paradigm into the object-oriented paradigm. However, since the object-oriented paradigm reflects the distributed structure of the problem space and is suitable for representing inherently concurrent behavior, and since the functional paradigm allows us to explore parallelism on the parallel processing system in the distributed sites, PROOF is also suitable to support the software development for large scale distributed parallel computing systems with the benefits of both the object-oriented and functional paradigms.

Our framework has the following phases: decomposition, object design, verification, coding, partitioning and allocation. In this paper, we will discuss the overall framework for the software development and focus on the decomposition, object design and verification and analysis phases. We assume that the requirement specifications are given and coding using the PROOF/L language [2] is a straightforward phase, and hence they will not be discussed here.

3 Decomposition

The decomposition phase is based on the object-oriented strategy and consists of the following steps:

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.

0-8186-2755-7/92 $03.00 © 1992 IEEE

7) Identify bottleneck objects, if any.

8) Check the completeness and consistency of the decomposition.

In Step 1), 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. One of the strategies to identify the objects is by examining the specification written in natural languages. The nouns in the specification can be candidates for the objects and the verbs as the operations of that object [4]. Another strategy is to draw the dataflow diagrams first and then detect the candidates for objects from this diagram [5]. Other techniques for identifying the objects are summarized in [4]. These techniques can be used as guidelines for identifying objects. However, the experience and intuition of the developers still play an important role in identifying the objects from the requirement specifications.

In Step 2), class interfaces are determined by identifying methods provided by each object class and then defining the inputs and outputs of those methods. The actual design of the methods is postponed until the object design stage.

In Step 3), the static relationships among objects are specified by identifying the methods required by each object. This relationship is specified using the object communication diagrams. The identity of the objects, the relationships among them and their methods are specified so that the features of the real world problem which are important for the software developer can be captured.

In Step 4), the objects are classified according to their invocation as *active, passive* and *pseudo-active*. An active object can initiate activation of other objects by invoking methods of other objects. A passive object is activated only when its methods are invoked by other objects. Pseudo-active objects behave between the purely active and purely passive objects. Pseudo-active objects can invoke the methods of other objects and has methods which can be invoked by other objects. Since active objects are invoked when the software system is started, all the threads of control in the application start from the active objects. Identifying all the threads is very important in real-time process control systems. We can identify all the possible threads of control and then use this information to check for the completeness and the consistency of the decomposition. Classification of objects by their invocation behavior helps to build the static structure of the software system among objects.

In Step 5), once the static structure of the software system is determined, we identify shared objects from them. Shared object has local data which can be accessed by a number of objects. The shared objects can be further divided into two classes of objects: *read-only* shared object and *writable* shared object. The read-only object has local data which cannot be modified by other objects or has no local data. 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. However, writable objects cannot be duplicated easily. All the access to the data in the writable objects needs to be synchronized to maintain the consistent status of the data. Shared writable objects could become bottleneck objects as they may have to be executed sequentially to maintain the consistency of the data. Such bottleneck objects are often shared components requiring synchronization among objects accessing it concurrently. Thus, identifying bottleneck objects from the decomposition and refining the decomposition to reduce the number of unnecessary bottleneck objects may play an important role in enhancing the parallelism.

In Step 6), the behavior of each object is specified. The object communication diagram obtained in Step 3) only describes the static structure and relationships of the objects in the problem domain. It does not provide any information regarding the behavior of the software system to be developed. That is, the control aspect of the software system is not specified in the object communication diagram. However, to verify and analyze the decomposition, we need to define the behavior of the objects. For this purpose, we use the notations similar to those in [6]:

- SEQuencial execution of methods: When the methods $m_1, m_2, \ldots, m_n$ are executed sequentially in the order $m_1, m_2, \ldots, m_n$, its behavior is specified as $\mathrm{SEQ}(m_1, m_2, \ldots, m_n)$
- CONcurrent execution of methods: When the methods $m_1, m_2, \ldots, m_n$ are executed concurrently, its behavior is specifed as $\mathrm{CON}(m_1, m_2, \ldots, m_n)$
- WAIT for method invocation: When an object is waiting for the invocation of its method m by another object O to proceed with its execution, its behavior is specified as $\mathrm{WAIT}(m, O)$
- SELect a method for execution based on a condition: SEL construct behaves like the CASE statement in ordinary programming languages. When an object selects one of the methods for execution from among the methods $m_1, m_2, \ldots, m_n$ based on a condition, its behavior is specified as $\mathrm{SEL}(m_1, m_2, \ldots, m_n)$
- ONE-OF the methods for execution from a group of possible methods: ONE-OF construct is used in cases where different objects could try to invoke the methods defined in the object O simultaneously. The object O permits only one object to invoke its method at a time. This construct serializes the requests and is typically used to describe the behavior of the shared writable objects. Note the difference between the SEL and the ONE-OF construct. Among the set of methods $m_1, \ldots, m_n$, defined in an object, when the object permits only one of its methods to be invoked by other objects, the behavior of the object is specified as $\mathrm{ONE-OF}(WAIT(m_1, O_j), \ldots, WAIT(m_n, O_k))$

In Step 7), the bottleneck object which may unnecessarily degrade the performance of the software system is identified. Usually, a bottleneck object will be a shared writable object. Such objects limit the parallelism in the software system. If such an object is found, then redo or refine the decomposition to reduce the bottleneck if possible. This step may increase the

number of objects now available in the software system. Repeat Steps 2) to 6) until the decomposition is found satisfactory.

In Step 8), the result of the decomposition is verified with the user requirements. From the given user requirements, the possible scenarios of activities are identified, and each of them is examined using the behavior of the objects specified in Step 5). The first activity in any scenario must begin in one of the active objects. The sequence of activities in each scenario must be reachable by tracing the behavior of the objects. If there is any scenario that cannot be followed, the decomposition is incorrect and the decomposition steps need to be reviewed. The consistency among objects is verified by examining whether input parameters of the methods being called are defined as local variables in the calling object and output parameters of the methods being called are defined as local variables in the called object.

4 Object Design

The object design is specified using the notations defined in PROOF/L [2]. The class interface definitions and information about the object behavior are used to design the objects. We have identified three steps in our approach to the object design:

1) Establish the class hierarchy.

2) Design the method .

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

Step 1) Because some common operations and/or attributes between the objects may not be apparent in the decomposition phase, different objects should be 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 some cases, a superclass can be extracted from a single subclass and put in the class library if needed. Establishing a class hierarchy in the form of superclasses and subclasses increases the inheritance in the application. Class hierarchy also increases the modularity of the software and enhances the extensibility of the software [7].

Step 2) 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 synchronization among concurrent objects is achieved by the guards attached to the methods. The expression is specified informally in a natural language. If there is a guard, the method is executed only if the guard is true; otherwise, the method waits till the guard becomes true. When there are simultaneous attempts to access the same object through invocation of its methods, the selection of one method for execution is done non-deterministically.

It is desirable to refine the methods that access the shared objects. For example, let object $O1$ invoke a method m defined in the shared object $O2$. Now suppose that the method m requires to read data, perform some computation based on the data and then modify the local data of $O2$. Then the guard of the method m needs to be evaluated before the execution of the method m begins. The activities, reading and computing, performed on $O2$ can be executed in parallel when another object invokes this method because those operations do not involve any shared data. However, these activities cannot be executed by another object in parallel if the method m contains these activities as part of its code. Thus, the method should be refined into smaller methods in such a way that the guard can affect the execution of a short segment of code only. This refinement of method is similar to the refinement of the object to reduce the bottleneck in the decomposition stage.

Algorithm and data structure selection is a significant part of the method design. The selection of algorithms to accomplish a specific task should be based on certain criteria which satisfy the required constraints such as accuracy, timing requirements, use of common utilities across the design, reuse of previously developed software, computational complexity, flexibility, ease of implementation, understandability, etc.

While designing the algorithms, new classes of objects may be defined to make the implementation more efficient. These are low level objects and are not usually visible externally.

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. The modification of objects is expressed using the special construct $\mathcal{R}$ as $\mathcal{R}[|O|]e$ in which O is the object called the *recipient object* that receives a new state obtained as the result of evaluating e [2]. The modification of the objects is allowed only at the bodies of the objects. Thus, there is no side effect in the method, and history sensitivity in the object level is achieved.

The body of an object can be derived using the class interface and the object behavior obtained from the decomposition stage. We also need to introduce the modification operator $\mathcal{R}$ in the body of the objects that are modified. The objects that are modified can be determined from the method definitions given in the class interface. Consider an object O_1 defined as the output of a method m. Whenever an object O_2 invokes the method m defined in O_1, O_2 will be modified. Thus, in the body of O_2, when m is invoked, the modification operator $\mathcal{R}[| O_2 |]\, m$ is substituted in the place of the method invocation.

5 Verification

The design of the objects done in the previous phase has to be analyzed for various *liveness* and *safeness*

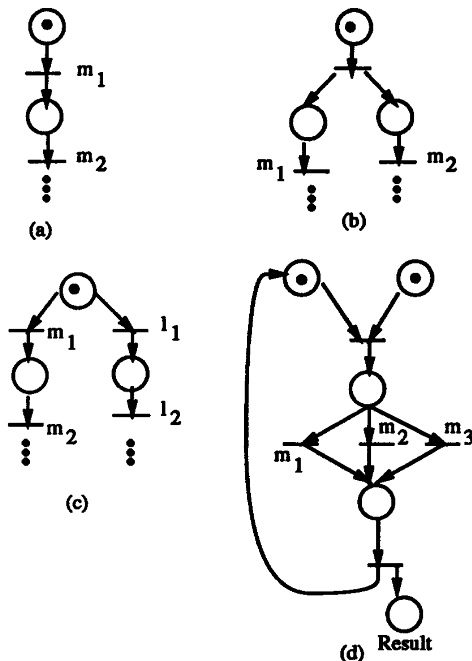


Figure 1: Transformation rules for the control constructs: (a) SEQ($m_1, m_2, \ldots$), (b) CON(SEQ($m_1, \ldots$), SEQ($m_2, \ldots$)), (c) SEL(SEQ($m_1, m_2, \ldots$), SEQ($l_1, l_2, \ldots$)), (d) ONE-OF(m_1, m_2, m_3)

properties. For this purpose, we transform our design into Petri Nets [8]. Petri nets have been selected in our approach mainly because our design can be easily represented in Petri net model and because many techniques have been developed to analyze Petri-net models for various liveness and safeness properties [9, 10].

The transformation of the design to Petri nets consists of the following three steps:

1) Transformation of bodies to Petri nets.

2) Composition of the nets.

3) Refinement of the nets.

Step 1) To transform the bodies into Petri nets, we use places as the token holder for the control flow, transitions as the methods, and the arcs between places and transitions as the control flows. Since a body is represented as a statement consisting of control constructs and method names, we show the transformation rules for each control construct in Fig. 1. Body of an active object could have methods that do not require modifications and methods that require modification by using the construct $\mathcal{R}$. The method requiring modification needs to be executed in serial to maintain the consistency of the object state. For this purpose, an additional place called the ***bottleneck*** place is associated with such a method so that the serialization of the execution can be specified. Fig. 2

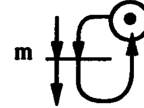


Figure 2: Transformation of a method requiring modification

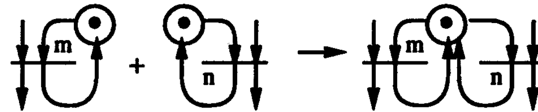


Figure 3: Combining two objects with a common bottleneck place

shows the transformation of such a method. The bottleneck place will also be used to compose the Petri nets in the next step.

Step 2) The Petri nets obtained from the bodies of the active objects in Step 1) must be composed together so that the software system can be represented by a single Petri net representation. To compose the nets, we need to identify the transitions or the places that serve as interaction points. The interaction among objects occurs only when there is an object modified by other objects. When an object interacts with another object for accessing the shared writable object, the bottleneck place will be common to both the objects. Since the bottleneck place is used to serialize the interaction among the methods requiring modification, they can be used as the fusion point. When the nets are to be composed, the body of the active object is searched for the methods that require modification. When such methods are found, two cases arise. For example, consider two active objects O_A and O_B having the following bodies:

O_A: SEQ($m_1, \ldots, \mathcal{R}[| O_i |] m_n$)
O_B: SEQ($m_{1'}, \ldots, \mathcal{R}[| O_j |] m_{n'}$)

When the methods m_n and $m_{n'}$ are defined in different classes ($O_i \neq O_j$) there is no common bottleneck place between O_A and O_B. Hence, no composition of the nets is necessary. When the methods m_n and $m_{n'}$ are defined in the same object ($O_i = O_j$), the two bottleneck places associated with the two methods are combined to one. This process is called ***fusion*** of places and is illustrated in Fig. 3.

Step 3) The purpose of the refinement is to replace a transition by a more complex Petri net in order to give a more detailed description of the activity involved in the transition. It is analogous to the module concepts found in many programming languages. At one level, a simple abstract description of the activity is given without considering the detailed behavior. At another level, by refining the nets, a more detailed description of the activities can be specified.

The transition can be refined according to the following rules. Suppose that a transition t_i is replaced with a subnet S. The subnet S consists of three parts: input transition, a refinement of net called block, and

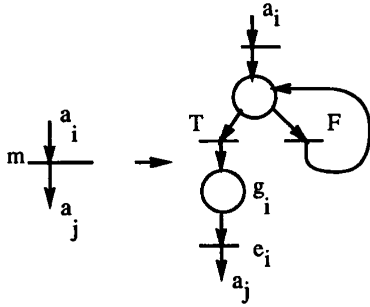


Figure 4: Refinement of the methods with the guards.

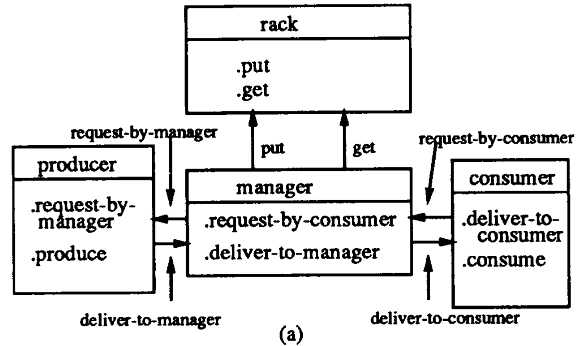
output transition. The incoming arcs of t_i serve as incoming arcs to the input transition. The outgoing arcs from t_i serve as outgoing arcs from the output transition. All the transitions except the input and output transitions can only interact with the places defined within the block of S. All the places can only interact with transitions defined within S.

Since a method consists of an expression with an optional guard, the transitions may have to be refined to specify the guard and the expression and is done as follows: Let a method m_i consist of a guard g_i and an expression e_i. Then the transition for m_i can be refined as follows: Guard evaluation is specified as a place and there is a transition associated with each result - true or false - of the guard evaluation. In case of True transition, the expressions are executed. In case of False transition, go back to the guard place to evaluate the guard again. The use of True and False transitions is analogous to the method specified in [8] to represent condition statement. This refinement process is shown in Fig. 4.

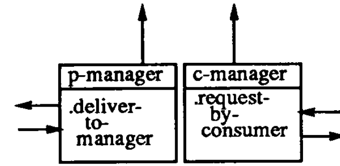
6 An Example

We will use the warehouse management system as an example. We will illustrate each step in our framework by taking only a couple of cases. A simplified version of the requirement of a warehouse management system is given below:
The consumer requests items from the warehouse manager and consumes the items delivered by the warehouse manager. The manager receives the order request from the consumer and then retrieves the requested items from the rack and deliver them to the consumer. The manager also places an order to the producer whenever needed. The producer produces items only when the manager requests them.

For the decomposition phase, we have :
1) The objects: *manager, producer, consumer,* and *rack.*
2) For the interface of the class *manager*, the inputs for the method *request-by-consumer* are *request* and *manager*; and its output is *manager;* the inputs for the method *deliver-to-manager* are *item* and *manager*; and its output is *manager.*



(a)



(b)

Figure 5: (a) An object communication diagram for the warehouse management system, and (b) object *manager* modified

3) Dependency and communication relationships among objects are identified and described using an object communication diagram shown in Fig. 5(a). The links between the objects indicate the method invocations between the objects. The arrows on the links indicate the direction of invocation.
4) *consumer* is an active object, *manager* is a pseudo-active object, *rack* and *producer* are passive objects.
5) *rack* is the only object with shared data. The methods defined in *rack* – *put* and *get* – update the local data in *rack*, and thus *rack* is a writable shared object.
6) The behavior of each object is specified. *rack* has two methods *put* and *get*, which are invoked by *manager*. The behavior of *rack* is specified as follows:
ONE-OF(WAIT(*put,manager*),WAIT(*get,manager*)).
consumer has two methods *deliver-to-consumer* and *consume* and its behavior is specified as follows:
SEQ(*request-by-consumer*,WAIT(*deliver-to-consumer,manager*), *consume*)
Similarly, we have the behavior of other objects:
producer: SEQ(WAIT(*request-by-manager, manager*), *produce, deliver-to-manager*)
manager: CON(SEQ(*request-by-manager*, WAIT(*deliver-to-manager, producer), put*), SEQ(WAIT(*request-by-consumer, consumer*), *get, deliver-to-consumer*))
7) In *manager* there are two threads of control: one is initiated by *manager* to receive the item from *producer* and store it to *rack* and the other is initiated by *consumer* and to retrieve the item and send it to *consumer.* Thus, the object *manager* can be split into two

objects since two independent threads of control exist in *manager*. This enhances parallelism. After splitting *manager* into two different objects, called *p-manager* and *c-manager* as shown in Fig.5(b), the new object communication diagram can be obtained by substituting Fig.5(b) in place of the *manager* in Fig. 5(a). The object behavior for each of the objects can be derived as follows:
rack: ONE-OF(WAIT(*put*, *p-manager*), WAIT(*get*, *c-manager*))
consumer: SEQ(*request-by-consumer*,WAIT(*deliver-to-consumer*,*c-manager*),*consume*)
producer: SEQ(WAIT(*request-by-manager*, *p-manager*), *produce*, *deliver-to-manager*)
p-manager: SEQ(*request-by-manager*,WAIT(*deliver-to-manager*,*producer*), *put*)
c-manager: SEQ(WAIT(*request-by-consumer*, *consumer*), *get deliver-to-consumer*)
8) The following control threads are identified from the requirements: (a) 'the consumer requests an item from the c-manager and the c-manager retrieves it from the rack and sends it to the consumer' and (b) 'the p-manager asks the producer to produce items and stores them to the rack'. These control threads can be verified by starting from the active objects, *consumer* and *p-manager*, and tracing the method invocations using the object behavior.

For the design phase, 1) Establish class hierachy. In this example, each object is an instance of a different class. Thus no class hierarchy exists based on the inheritance. However, suppose that *p-manager* and *c-manager* may need to record all the transations done in each object. Then both objects will need a method 'update-transaction-record'. In such a case, we can define superclass, called *manager-class* having the method update-transaction-record, that can be inherited to its subclass, *p-manager* and *c-manager*.
2) Design of methods. Since an object is an instance of a class, we define the class. In the following, we only give the definition of the class *Rack*. The others can be done in a similar manner.

```
class definition Rack(itemtype, capacity)
   -- rack is defined as generic class with
   -- itemtype and capacity as parameters.
 composition
   buffer: list of itemtype X count:integer;
   -- buffer is a storage for items and count
   -- indicates number of items in buffer
 method put buffer x
      guard: count < capacity;
      expression
         add an item x to the buffer
 method get buffer
      guard: count > 0
      expression
         retrieve an item from the buffer
end class
```

3) Determine the body of active/pseudo-active objects. We have two active/pseudo-active objects, *p-manager* and *consumer*. In the following, we show how the body of *p-manager* can be obtained from the bahavior of the objects.

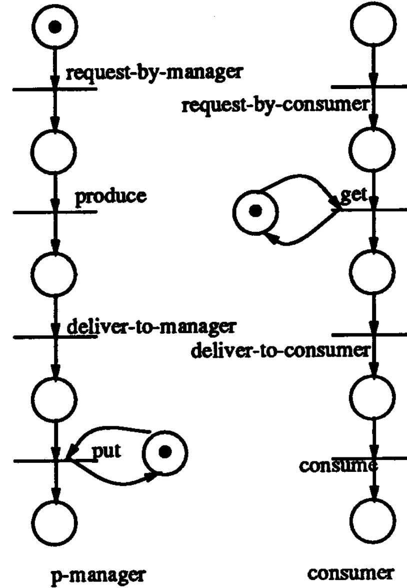


Figure 6: Transformation of the bodies p-manager and consumer.

An important aspect of the design here is indicating the modification of the objects. For this we attach $\mathcal{R}$ at the method *put* since this method modified the p-manager. Thus, $\mathcal{R}$[|*rack*|] *put* is substituted in place of *put*. This modification of body will be implemented at the coding stage. The design of other constructs is straightforward and will not be discussed here.

Verification is done by transforming the design into Petri-nets and then applying one of the many Petri-net algorithms to verify the net for deadlock, livelock, etc. We will consider the bodies of the active objects *p-manager* and *consumer* which are given respectively as follows:

SEQ(request-by-manager, produce, deliver-to-manager,R[|*rack*|]*put*)

SEQ(request-by-consumer, R[|*rack*|]*get*, deliver-to-consumer, consume)

In Step 1), each body is translated. For each body, based on the control structures used, the corresponding net structure is copied and methods are associated with the transitions. In the case of the methods that are attached to R, the bottleneck place is added to the corresponding transition. This step is shown in Fig. 6. In Step 2), the two bottleneck places are fused to compose the two nets into one. Because the methods put and get are from the same object, we fuse the two nets into one via the bottleneck place. In Step 3), the transitions representing methods with guards are also refined resulting in the final net. Because the methods put and get have guards, the net is refined to describe

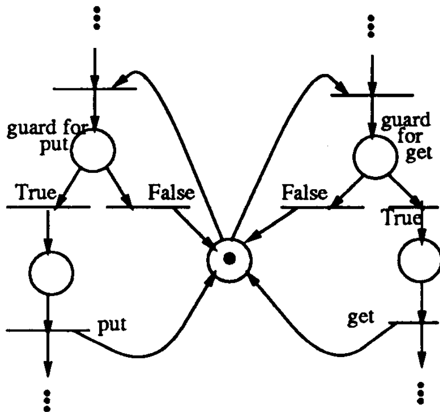


Figure 7: Refinement of the methods with the guards.

these guards. This is shown in Fig. 7. Various Petri Net based algorithms can now be applied to this net to analyze the net for the desired properties.

7 Discussion

In this paper, we have presented a framework for the software development for distributed parallel processing systems without considering the network details which will be handled by the translator in the implementation. The implementation phase is done in the following steps: *coding, translating to a target code, partitioning, allocation* and *execution*. Coding is perfomed after the verification stage. Coding in our framework will be done using the PROOF/L language which is based on PROOF. The advantage of using this language is that the programmer will not have to be bothered by the synchronization, communication or the configuration of the network. PROOF/L code is translated into the target code and then executed on the target machine. In this case, the target machine will be the network of parallel computers. Translating PROOF/L into the target language can be done in two steps: In the first step, PROOF/L code is transformed into an intermediate form. This intermediate form will be independent of any particular language and will not reflect any characteristics of any specific architecture. This will facilitate the implementation of the PROOF/L code on different computers and different target language easily. Also, the intermediate form will facilitate the exploration of the fine grained parallelism. The second step is to translate the intermediate form into the target language. The architecture of the target computer as well as the features of the target language will have to be used as inputs for this translation step.

Partitioning is performed to reduce the communication overhead among objects by clustering together the objects that communicate extensively with each other. Partitioning can be done after the verification stage. The inputs to this stage are the object communication diagram, the number of times a thread of control is invoked and the number of times certain operations are performed. This information based on heuristics helps in estimating the number of times a method is invoked in an object. Based on this information, the communication overhead between objects can be determined and used by the partitioning algorithm. The partitioned cluster could be allocated to a site in the distributed system. Each method in an object can be the unit of parallelism that can be executed in the parallel processing system in a site.

The intermediate form, architectural details and the target language details are used as inputs to produce the target code from the intermediate form. The partitioned and translated code will be allocated onto the various processors in the network for execution.

References

[1] David B. Skillicorn, "Architecture-Independent Parallel Computation," *IEEE Computer,* Vol. 23, No. 12, December 1990, pp. 38–50

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

[3] 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 International Computer Software & Applications Conference,* (COMPSAC 91), September 1991, pp. 453-458.

[4] G. Booch, "Object-Oriented Development," *IEEE Trans. on Software Engineering*, Vol. SE-12, No. 2, Feb. 1986, pp. 211–221.

[5] S.C. Bailin, "An Object-Oriented Requirements Specification Method," *Communications ACM,* Vol. 32, No. 5, May 1989, pp. 608-623.

[6] S.S. Yau, C.-C. Yang, and S.M. Shatz, "An Approach to Distributed Computing System Software Design," *IEEE Trans. on Software Engineering*, Vol. SE-7, No. 4, July. 1981, pp. 427–436.

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

[8] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, 1981.

[9] S. S. Yau and M. U. Caglayan, "Distributed Software System Design Representation Using Modified Petri Nets," *IEEE Trans. on Software Engineering*, Vol. SE-9, No. 6, Nov. 1983, pp. 733–745.

[10] S. S. Yau and C.-R Chou, "Control Flow Analysis of Distributed Computing System Software Using Structured Petri Net Model," *Proc. Workshop on the Future Trends of Distributed Computing Systems in the 1990s*, Sept. 1988, pp. 174–183.