IMPLEMENTATION OF SYNCHRONIZATION PROTOCOLS

Master Thesis

Candidate: Daniel Mikušík
Advisor: František Plášil

Prague, April 1998
Acknowledgement

I offer my sincere gratitude to my advisor and instructor Dr. František Plašil for his dedication in guiding this work. His rich experience in the field, without which this thesis would not have materialized, helped me facilitate my progress in this work and aroused my interest for further research.
I declare that I have elaborated this master thesis on my own and listed all used references.

Prague, 25th April 1998
Mouo:

Everything should be done as simply as possible, but not too simple.

Albert Einstein
CONTENTS

1. INTRODUCTION .................................................................................................................. 2
   1.1 Synchronization in Concurrent OO Languages .......................................................... 2
   1.2 Sound Incremental Modification of Inherited Synchronization Policies .................... 3
   1.3 Goals of the Thesis ...................................................................................................... 4

2. ENRICHABLE PROTOCOLS ............................................................................................... 5
   2.1 Protocol Syntax ......................................................................................................... 5
   2.2 Protocol Semantics ..................................................................................................... 6
   2.3 Representation of Protocols ...................................................................................... 8

3. PROTOCOL CONTROLLER .............................................................................................. 10
   3.1 The Role of a Protocol Controller ............................................................................ 10
   3.2 Queue of Requests ................................................................................................... 11
   3.3 Protocol Controller Illustrative Example of Functionality ....................................... 12

4. DYNAMIC PROTOCOL TRACING ................................................................................... 14
   4.1 Protocol Threads ...................................................................................................... 14
   4.2 Initialization of a Protocol Controller ....................................................................... 19
   4.3 Actions Performed upon Method Invocation ............................................................ 21
   4.4 Actions Performed upon End of Method Servicing .................................................. 23
   4.5 Management of Guards ............................................................................................ 25

5. PROTOCOL INHERITANCE .............................................................................................. 27
   5.1 Enrichment Rule ....................................................................................................... 27
   5.2 Employing Enrichment Rules in Incremental Modification of Protocols .................. 31
   5.3 Guards during Protocol Enrichment .......................................................................... 32
   5.4 Issues Encountered Relating Protocol Enrichment .................................................. 34
      5.4.1 Representation Dependent Enrichment .............................................................. 34
      5.4.2 Guards in the Original Protocol .................................................................... 34
      5.4.3 Context Sensitivity ......................................................................................... 35

6. OPEN PROBLEMS, POTENTIAL FUTURE WORK ............................................................ 36

7. SUMMARY INSTEAD OF A CONCLUSION .................................................................... 38

8. REFERENCES ..................................................................................................................... 40
1. Introduction

1.1 Synchronization in Concurrent OO Languages

In procedural programming languages supporting concurrency, a need for synchronization tools based on the data type abstraction was recognized in the early seventies (e.g. monitors [Hoa74], path expressions [CH74]). With the spreading of object-oriented (OO) languages, a lot of effort was devoted to combining concurrency with the OO paradigm leading to the notion of Object-Oriented Concurrent Programming Languages (OOCPL). Of all the options they offer, we will limit ourselves to passive objects visited by multiple threads (potential concurrent invocation of an object's methods). Numerous synchronization mechanisms for this purpose have been designed during past decades.

Most of the existing mechanisms are based on the idea to represent an object's synchronization state and its history by the object's attributes and the state of the processing of the object's synchronization code. In a more acceptable case, such a mechanism separates strictly the "synchronization" and "sequential" attributes, and/or also the synchronization and sequential codes. This also provides a significantly better level of modularity. The type of a synchronization attribute may be at a high level of abstraction, like enabled-sets [TS89] or a reference to an object of the behavior class type [Atk90], or may be at a low level of abstraction, e.g. an integer or enumeration type. The state of the processing of an object's synchronization code is captured by the underlying implementation. However, parts of the state may be accessible in the synchronization code, for example, as scheduling predicates or synchronization counters. Naturally, the form of the synchronization code must comply with the operations defined for the types of synchronization attributes used in a particular object. These operations are evaluated as reactions to specific events, typically arriving of a method's execution request, finishing of the execution etc., or are even explicitly called in the sequential code of the object's methods. In general, there are many different notations for expressing synchronization code. To the representative ones belong guards (in a number of variants, e.g. [MWY90], [Ada94], [BLR94], [Fro92], synchronizers [MY93], protocols (Path expressions [CH74], PROCOL [BL91], [Flo95]), enabled-sets, mediators [GC86], and generic synchronization policies [CMH95].

The expressive power of a synchronization mechanism is usually judged only informally by implementing several well-known synchronization policies. More objective criteria are provided by Bloom in [Blo79]: A synchronization mechanism has a good expressive power if it has access to the following six types of information: the name of the invoked operation, the relative arrival time of invocation, the actual parameters of the invocations, the synchronization state of the object, instance variables, and the history (of finished invocations).

First articulated in ([MWY90], [MY93]), one of the key issues of tying the OO paradigm and concurrency is the difficulty of combining inheritance and synchronization. The issue is
closely related to the inheritance anomaly problem; for a detailed discussion we refer the 
reader to [MY93], [Mes93], [KL90], [CMH95]. In this paper, supporting McHale's 
opinion [CMH95] that the problem is intrinsic to inheritance rather than being a conflict 
between synchronization and inheritance, we follow his suggestion to tackle the problem 
by designing a new inheritance model or alternative ways to reuse code, rather than 
designing a new synchronization mechanism.

During inheritance, the synchronization code should be subject to incremental changes to 
reflect incremental modifications in synchronization policy. However, there are very few 
synchronization mechanisms, which allow such incremental changes. Enabled-sets, 
Sos/Esp [CMH95], and activation conditions in Guide [BLR94] are representative 
examples. To our knowledge, as for the protocol-based synchronization mechanisms, little 
effort has been devoted to incremental modification of protocols during inheritance. In our 
view, a protocol-based synchronization mechanism is very powerful, particularly with 
respect to providing information on synchronization status and history. Even though we 
understand that to achieve what [CMH95] calls "degrade gracefully" (or to avoid 
"creeping featureism" [CMH95]), the protocol idea has to be combined with some other 
synchronization tools, e.g. event handlers (actions) and synchronization variables 
[CMH95], we believe that the protocol paradigm is worth cultivation as it provides a 
remarkable ease of expression of many of the standard synchronization policies.

1.2 Sound Incremental Modification of Inherited Synchronization Policies

Another important issue is the quality of incremental modification of synchronization 
policies during inheritance. Instinctively, such a change should not be "upside-down"; for 
example, while operating upon a particular object o, to replace the requirement of 
sequential execution of methods o.a and o.b by allowing their parallel execution is 
intuitively wrong. On contrary, to add a new method c and require o.c to be executed in 
a particular order with o.a and o.b is intuitively appropriate. In the rest of the paper, we 
refer to such an intuitively appropriate modification as sound modification.

The key issue is how to check the soundness of an incremental modification. To our 
knowledge, the only attitude in reflecting the idea of soundness is the requirement of 
strengthening guards (e.g. activation conditions in Guide, synchronization constrains in 
[Fro92]). Naturally, this approach is limiting as it fails, for example, in the case of very 
simple history sensitivity: o.c is to be executed after o.a and, in a subclass, o.c is to be 
executed after o.a or o.b where b is a new method of the subclass. It is apparent that 
"oring" of guards would work in cases like this. In our view, the policy of strengthening 
guards can be interpreted as an effort to find an analogy with the policy of controlled 
modification of a redefined method's precondition and postcondition in Eiffel [Mey92] 
(strengthening postconditions, weakening preconditions in subclasses).
1.3 Goals of the Thesis

The goal of this thesis is to develop algorithms needed to implement Enrichable Protocols as part of a functional synchronization mechanism. Firstly, the protocol controller algorithm has to be developed. It must be able to decide whether an incoming request may be executed or must be put in a queue. The protocol controller must monitor the synchronization state and allow the queued requests to execute as soon as the protocol semantics allows to do so. The Enrichable Protocol semantics has to be defined in detail as it has been done only intuitively [PG95]. As we want to allow only sound modification of protocols, an algorithm testing for soundness has to be developed as well. Finally, the rule application algorithm has to be implemented to enable protocol inheritance.
2. Enrichable Protocols

This section describes the concept of synchronization protocols. Synchronization protocols are used to define synchronization constraints on execution of methods in an object.

2.1 Protocol Syntax

Our protocol definition is based on the definition of Enrichable Protocol defined by context-free grammar in [PG95]. The first protocol-based synchronization mechanism was originally introduced in PROCOL [BL91] as a way to separate synchronization from method bodies.

\[
\begin{align*}
G &= \{N, T, P, A\}; \\
N &= \{A, B, C, D, E, F, M, G\} \\
T &= \{+, :, |, *, ^, (, ), [ , ] \} \\
P &= \{ \\
A &\rightarrow B , \quad A \rightarrow A+B, \\
B &\rightarrow C, \quad B \rightarrow B; C, \\
C &\rightarrow D, \quad C \rightarrow C | D, \\
D &\rightarrow E, \quad D \rightarrow E^*, \quad D \rightarrow E^+, \\
E &\rightarrow F, \quad E \rightarrow [G] F, \\
F &\rightarrow M, \quad F \rightarrow (A) \\
M &\rightarrow //C++ expression \\
G &\rightarrow //C++ identifier
\end{align*}
\]

where:
- \(G\) - guard
- \(M\) - method expression
- \(+\) - alternative
- \(\;\) - sequencing
- \(\mid\) - potential parallel execution
- \(*\) - repetition via sequencing \(p^* = (p ; p ; \ldots ; p)\)
- \(^\wedge\) - potential re-entrant execution \(p^\wedge = (p \mid p \mid \ldots \mid p)\)
Example of a valid protocol:

\[(\text{fnct}_1 + \text{[guard]} \text{fnct}_2) \; ; \text{fnct}_1\]

### 2.2 Protocol Semantics

Protocols are used to specify restrictions on execution of object methods. Before we can develop an algorithm to enforce these restrictions, the exact semantics of protocols has to be defined.

Let us consider the following protocol \(P\):

\[
\text{method}_a
\]

It is derived from grammar \(G\) by application of rules:

\[
A \rightarrow B, \; B \rightarrow C, \; C \rightarrow D, \; D \rightarrow E, \; E \rightarrow F, \; F \rightarrow M, \; M \rightarrow \text{method}_a
\]

If the above protocol \(P\) is synchronizing an object \(O\), the only method that can be executed is \(O.\text{method}_a\). When the call to \(\text{method}_a\) arrives and the execution is over, we say that protocol \(P\) has **finished**, because there is nothing left to execute in the object.

Let \(X\) and \(Y\) be protocols. \(P\) is defined as \((X) \; + \; (Y)\) \((P = X \; + \; Y)\). \(P\) is also a valid protocol because \(A \rightarrow* (A) \; + \; (A)\) according to grammar in (2.1). When the protocol \(P\) is guarding an object, either \(X\) or \(Y\) may execute. \(P\) will finish as soon as \(X\) or \(Y\) finishes depending on which one of them was actually executed.

**Example 1:**

\[P = \text{method}_a + \text{method}_b\]

In this example either \(\text{method}_a\) or \(\text{method}_b\) can be run. If the call to the \(\text{method}_b\) comes and its handler has finished executing, the protocol \(P\) has finished.

Let \(P = X \; ; \; Y\).

After the protocol \(X\) is finished, protocol \(Y\) can be executed. \(P\) is finished as soon as \(Y\) is finished.

Let \(P = X \; | \; Y\).

Protocols \(X\) and \(Y\) may be executed in parallel, i.e. \(Y\) can be executed while \(X\) is running and vice versa. Both \(X\) and \(Y\) may be executed at most once. Suppose that both \(X\) and \(Y\) are running and \(Y\) finishes running first. In this case \(P\) finishes as soon as \(X\) finishes running. If on the other hand \(X\) was running and finished without \(Y\) being executed, \(P\) finishes - which means that \(Y\) can not execute any more.
Example 2

\[ P = \text{method}_a \mid \text{method}_b \]

Scenario 1:
Suppose that a call to \text{method}_a came. While \text{method}_a was running, a call to \text{method}_b came. \text{method}_b was executed and was running in parallel with \text{method}_a. At this point, no other calls would be accepted. Even after \text{method}_b has finished (\text{method}_a was still running), another call to \text{method}_b can not be accepted. Protocol \( P \) has finished as soon as \text{method}_a finished servicing its call.

Scenario 2:
Suppose that a call to \text{method}_a came. While it was running no other call came. Protocol \( P \) has finished as soon as \text{method}_a finished servicing its call.

Let \( P = X^* \)

This protocol specifies that \( X \) can be executed many times in sequence (after one execution of \( X \) finishes, another can execute), or it may not be executed at all (\( P \) will immediately finish).

Let \( P = X^\triangledown \)

\( X \) can be executed many times in parallel. When all the calls finish executing (there is no other call to \( X \) running), \( P \) finishes.

Let \( P = [\text{guard}_a] \ X \)

\( X \) can execute only if \text{guard}_a() evaluates to TRUE at time of the request of \( X \). Management of guards will be discussed in more detail in section 4.5.
2.3 Representation of Protocols

A protocol is represented as a syntactical binary tree consisting of the ProtNode objects:

```cpp
class ProtNode {
    char NodeType;    // Distinguishes between +, ;, |, ^, *,
                      // F(unction)
    int NodeId;       // Unique identifier in the scope of all the protocol
    int TypeId;       // Unique identifier in the scope of given
                      // NodeType
    ProtNode *Up;     // Points to the parent of this node in the
                      // protocol tree
    ProtNode *Left;   // Left branch of the protocol tree
    ProtNode *Right;  // Right branch of the protocol tree
    ProtNode *NextList; // Used to connect all the nodes in a linear list
    GuardList *Guards; // Points to the list of guards of this node
}
```

The `Left` pointer points to the left hand side of an operator, the `Right` pointer points to the right hand side of an operator. In case of unary operators (*, ^) the `Right` pointer is NULL. The `Up` pointer is present to enable stepping through the protocol tree in both directions.
Figure 1 shows a sample protocol representation:

\[
[glob\textunderscore guard]\text{(method}_a + [guard}_b\text{ method}_b)^
\]

![Diagram of protocol representation]

**Figure 1  Sample Protocol Representation**

The **NodeId** identifier is unique protocol node identifier (i.e. there are no two different nodes with the same **NodeId**). It is assigned values starting with 1 with increments of 1 as the protocol parser parses the protocol from left to right. The **TypeId** is unique in scope of one **NodeType**. In present time it is used only to distinguish between functions (nodes of type ‘F’).
3. Protocol Controller

3.1 The Role of a Protocol Controller

The protocol controller is an object responsible for synchronizing calls to the controlled (server) object. Its decisions are based on a supplied protocol. We can imagine the protocol controller as being in between of the client and the actual implementation. It is transparent in a way that both the client and the object implementation do not know about its presence. The client thinks that it is calling the implementation directly. Figure 2 shows a protocol controller with references to an object it should control, and to a protocol that tells it how to control the object.

![Diagram](image.png)

**Figure 2  Role of the Protocol Controller**

In order to execute an object method, a client should call the protocol controller and ask for execution of this method. When such a request arrives, the protocol controller has to check the state of the relating protocol, and execute the method if the protocol allows the call to be handled immediately. If the called method is blocked by the protocol, the request will be inserted into a queue of yet unhandled requests and will be serviced as soon as possible.
The protocol controller’s interface consists of three methods:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProtServer(char *Protocol)</td>
<td>Constructor which takes the Protocol as a parameter. It also parses the Protocol into its inner representation (see 2.3).</td>
</tr>
<tr>
<td>Request *Run(int MethodID)</td>
<td>Tells the protocol controller that we want to execute a method identified by MethodID that is supplied as the parameter. The server assigns the request unique identifier and returns it to the caller.</td>
</tr>
<tr>
<td>Finished(Request *r)</td>
<td>Notifies the protocol controller that a method has finished executing. The parameter is used to specify which method has finished executing. The value of r was obtained from the Run call.</td>
</tr>
</tbody>
</table>

### 3.2 Queue of Requests

As mentioned in the previous paragraph, the execution of an object method is controlled by the protocol controller. It uses a queue of RunInstances to identify and store the requests:

```cpp
class RunInstance {
    int FnctId; //unique identifier of requested method
    int IsRunning; //TRUE - the request is being serviced;
                   //FALSE - the request is blocked
    RunInstance *Next; //pointer to the next request
};
```

Every valid request is put into this queue. If the requested method is not blocked, the server stores TRUE to the request's IsRunning attribute and executes the handler.

When the state of the protocol changes (e.g. one handler finishes servicing a request), the protocol controller traverses the queue in FIFO order and executes handlers for all the requests that are not blocked any more and are not yet running (have FALSE in their IsRunning attribute).

The finished requests are removed from the queue.
3.3 Protocol Controller Illustrative Example of Functionality

Let us show the functionality of a protocol controller on a model example. A shared memory cell server provides two methods:

\[ \text{Set(value)} \quad \text{- sets value of a shared memory cell} \]

\[ \text{value Get()} \quad \text{- gets value of a shared memory cell} \]

The following restrictions on execution of the methods are defined: \( \text{Set} \) has to be called first to initialize the cell. After the cell is initialized, various clients may execute \( \text{Set} \) or \( \text{Get} \) any number of times. \( \text{Set} \) can not run in parallel with \( \text{Get} \). \( \text{Get} \) can be executed by more threads concurrently.

We can encode these restrictions by the following protocol:

\[ \text{Set} ; ( \text{Get}^* + \text{Set} )^* \]

The following table will illustrate the protocol controller functionality on a sample sequence of client requests as they arrive to the server.

<table>
<thead>
<tr>
<th>Request #</th>
<th>Requests and returns</th>
<th>Protocol controller actions</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Get</td>
<td>Request 1 queued</td>
<td>( \text{Get} ) can not execute now as first request has to be ( \text{Set} ).</td>
</tr>
<tr>
<td>2</td>
<td>Get</td>
<td>Request 2 queued</td>
<td>( \text{Get} ) can not execute now as first request has to be ( \text{Set} ).</td>
</tr>
<tr>
<td>3</td>
<td>Set(88)</td>
<td>Request 3 (Set) executed</td>
<td>( \text{Set} ) can execute. The first ( \text{Set} ) of the protocol (the one before ;) was executed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Request 3 (Set) has finished executing</td>
<td>Cell's value is now 88.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Request 1 (Get) executed from queue</td>
<td>The protocol controller looked into the queue of requests and selected first request that can execute ( in FIFO order)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Request 2 (Get) executed from queue</td>
<td>Although request 1 is still running, another ( \text{Get} ) may execute in parallel.</td>
</tr>
<tr>
<td>1</td>
<td>88 returned</td>
<td>Request 1 (Get) has finished executing</td>
<td>The first request to ( \text{Get} ) has finished executing and returned 88 to the client.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>4</td>
<td>Set(55)</td>
<td>Request 4 (Set) queued</td>
<td>Set can not execute now as the second Get is still running and Set can not run in parallel with Get.</td>
</tr>
<tr>
<td>5</td>
<td>Get</td>
<td>Request 5 (Get) executed</td>
<td>Two Gets (2 and 5) are now executing in parallel.</td>
</tr>
<tr>
<td>2</td>
<td>88 returned</td>
<td>Request 2 (Get) has finished executing</td>
<td>Request 2 has returned 88 to client. Request 5 is still running.</td>
</tr>
<tr>
<td>5</td>
<td>88 returned</td>
<td>Request 5 (Get) has finished executing</td>
<td>Request 5 has returned 88 to client although it was called after Set(55).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Request 4 (Set) executed from queue</td>
<td>There is no more Get running, so Set may execute.</td>
</tr>
<tr>
<td>6</td>
<td>Get</td>
<td>Request 6 (Get) queued</td>
<td>Get can not run with Set in parallel.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Request 4 (Set) has finished executing</td>
<td>Value of the cell is set to 55.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Request 6 (Get) executed from queue</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>55 returned</td>
<td>Request 6 (Get) has finished executing</td>
<td>55 was returned to the client.</td>
</tr>
</tbody>
</table>

The server calls may be synchronous or asynchronous. If the above server was called synchronously, it is obvious that it had to be called by multiple clients. Single client would wait forever for return from the first Get request.

The following section will show, how does the protocol controller make decisions about queuing and executing server requests.
4. Dynamic Protocol Tracing

This chapter will describe the mechanism of protocol tracing that is used during the lifetime of a protocol controller. Its purpose is to be able to tell at any given time, whether an incoming request may or may not be handled.

4.1 Protocol Threads

The protocol controller has to make decisions, whether a specific request may be handled or not. In order to do it, it holds a list of pointers to the protocol tree to identify nodes that are currently running and the ones that can execute if such a request to execute them arrives. Sometimes, however, it can not deterministically decide, which nodes in the protocol tree were executed, because the same method could execute in more than one node of the protocol tree. As the server doesn't know which requests will come in the future, and therefore can not eliminate any of these possibilities, it has to hold information about all of them. A place to store information about one such possibility is called a protocol thread.

Let us show this on an example:

\[ \text{A} | (\text{A;B}) \]

If a call to A arrives, we know that it is possible to service it, but we don't know which of the two A's to use. In a case like this we say that the first A was executed in a context of one protocol thread (thread1) and the second A was executed in another protocol thread (thread2). Hence the protocol controller has to keep track of two threads: the first one will mark that the first A was executed, the second one will mark that the second A was executed. If a call to B arrives after A is serviced, the server has to look whether there is such a thread in scope of which B could execute. In our case it can execute B in the context of thread2 only. As it wouldn't be able to service the call to method B in scope of the first thread (the protocol has already finished), the server will erase the first thread.

The protocol thread is a data structure representing one possibility of tracing a protocol. It contains information about running methods, methods that can be executed, and a reference to a Context structure that is holding other information needed to trace the protocol. New protocol thread objects are dynamically allocated as needed. Unusable threads are deleted.
class ProtThread {
    RunStatus *CanRunTbl[FNCT_COUNT];
    // Table of methods that can be executed
    RunQueue *RunningList;
    // Pointer to a queue of running methods
    Context *ContTree;
    // Contains information about a state of protocol nodes
    // with respect to tracing of a protocol (described in the
    // next chapter)
    ProtThread *Next;
    // Links protocol threads in a list
}

The Context structure contains status information about protocol nodes (ProtNode) of these types: ‘*’, ‘|’ and ‘^’: class Context {
    RecurStatus *Parent;   // Pointer to information about ‘^’ node in
    // scope of which is this context running
    SeqStatus *SeqList;    // Information about ‘*’ nodes
    ParalStatus *ParallList;  // Information about ‘|’ nodes
    RecurStatus *RecurList;  // Information about ‘^’ nodes
    GuardStatus *GuardList;  // Information about guards
    Context *Next;
}

In case of type ‘*’ we need to know whether some method has been executed in the underlying protocol:
class SeqStatus {
    ProtNode *Seq;          // Points to the corresponding node
                          // in the protocol tree
    int WasRanBefore;      // 0 - no, 1 - yes
    SeqStatus *Next;       // Pointer to the next SeqStatus in a queue
}

In case of type ‘|’ we need to know weather left and right branches are running, not running or have finished running:

class ParalStatus{
    ProtNode *Paral;       // Points to the corresponding node
                          // in the protocol tree
    short LeftStatus;      // NOT_RUNNING, RUNNING, FINISHED
    short RightStatus;     // NOT_RUNNING, RUNNING, FINISHED
    ParalStatus *Next;     // Pointer to the next ParalStatus in a queue
}

The ‘^’ operator allows us to execute the underlying protocol finite number of times in parallel. For every such execution we need to hold the Context information. For performance reasons we also keep the number of concurrently running protocols.

class RecurStatus {
    ProtNode *Recur;       // Points to the corresponding node
                          // in the protocol tree.
    Context *Parent;       // Context information in scope of which is this
                          // node located.
    short RunningCount;    // Number of protocols running in scope
                          // of this node.
    Context *ContList;     // List of contexts for every protocol running
                          // in scope of this node.
    RecurStatus *Next;     // Pointer to the next RecurStatus in a queue.
}
Other data structures pointed by attributes in ProtThread are RunStatus and RunQueue. RunStatus contains information about methods that can be executed and RunQueue contains information about methods currently running.

class RunStatus {
    ProtNode *What;  // Node of type ‘F’ which can be executed.
    Context *Where;  // Context in scope of which can this node
        // be executed.
    RunStatus *Next;  // Next possible execution information.
};

class RunQueue {
    RunInstance *Key;  // Pointer to the request queue to identify the request
        // being handled.
    ProtNode *What;  // Protocol node of type ‘F’ being handled.
    Context *Where;  // Context in scope of which is the handler (method)
        // running.
    RunQueue *Next;  // Next running information in a queue.
};

Let us demonstrate the relations between these structures on a simple example:

Protocol \{A | B^\} is used to synchronize an object that contains methods A and B. An
instance of a protocol controller is constructed and three requests: B(), A(), and B()
arrived to the server resulting in the following memory image:
Methods A and B are identified by 0 and 1 respectively. That is why TypeId of the first ‘F’ node corresponding to A is 0 and TypeId of the second ‘F’ node corresponding to B is 1. The picture shows only one protocol thread as all the choices to execute called method were deterministic. The protocol thread’s CanRunTbl contains two pointers (0 - A, 1 - B). The first one is NULL as we can not execute method A because it is already running. The second one points to the list of RunStatus structures showing under which context and where in the protocol tree could the method B execute. In our example, there is only one possibility.
The queue of requests contains three RunInstances because three requests arrived to the protocol controller (B, A, B). RunningsList contains also three instances as all of the requests are being handled.

The usage of the Context structure will be described in the following chapters, but let us now give a list of information it contains:

The ContTree pointer from ProtThread points to the top level Context structure. Protocol nodes with id's 0, 1 and 3 belong to the scope of this Context. The ParallList points to a list of ParalStatus structures. In our case there is only one node of type '|' so the list contains only one member. Both LeftStatus and RightStatus are equal to RUNNING which means that both left and right branches of protocol node 1 (NodeId=1) are currently running. The RecurlList pointer points to information about nodes of type '^'. It tells us that the tree under this node was executed twice (RunningCount=2) and the ContextList pointer points to the list of Contexts for these executions. Nothing is running under the rightmost Context, but it is there to identify "empty space" for another call of method B. Note that a member of CanRunTbl points to it.

### 4.2 Initialization of a Protocol Controller

Before the protocol controller can manage client requests it has to be initialized by its constructor. The protocol controller constructor receives a reference to a protocol tree and an object.

The constructor creates one protocol thread object with CanRunTbl [] entries and RunningList initialized by NULL. One Context object referenced by ContTree is created as well. All of its pointers are initialized by NULL. The class Context contains method WhatCanRun which steps through the protocol tree, fills the ProtThread's CanRunTbl with information about methods allowed to execute, and fills the Context's attributes. It takes one parameter poining to a protocol sub tree we want to process. The entire protocol (it's root) is used as a parameter to the WhatCanRun method to initialize the protocol controller.

```cpp
void Context::WhatCanRun(ProtNode *Here)
{
    if (Here->Guards) {
        GuardStatus *Temp = GuardList;
        GuardList = new GuardStatus(Here, Temp);
    }
    switch (Here->NodeType) {
    case '+' :
        WhatCanRun(Here->Left);
        WhatCanRun(Here->Right);
    ```
break;
    case '|':
        ParalList = new ParalStatus(Here, ParalList);
        WhatCanRun(Here->Left);
        WhatCanRun(Here->Right);
        break;
    case ';':
        WhatCanRun(Here->Left);
        break;
    case '*':
        // Fork the current protocol thread.
        ForkedThread = new ProtThread(this_thread);
        // execute Finish in the forked thread
        ForkedThread->Finish(Here);
        // Execute WhatCanRun in the current thread.
        SeqList = new SeqStatus(Here, SeqList);
        WhatCanRun(Here->Left);
        break;
    case '^':
        RecurList =
            new RecurStatus(Here, this, RecurList);
        break;
    case 'F':
        // Inserts the node Here along with this context
        // into the protocol thread's CanRunTbl.
        ActivePT->CanRunTbl[Here->TypeId] =
            new RunStatus(Here, this,
                ActivePT->CanRunTbl[Here->TypeId]);
        break;
    }
};

After this method finishes running, the protocol controller is ready to receive client requests. It contains one or more ProtThread structures holding enough information to know which requests may be handled and how to continue if the requests actually arrive. The reason why it may contain more than one protocol threads even if nothing was executed yet is the node of type `*`: the semantics of this node says, that it may execute it's sub tree zero or more times in sequence, so the decision is not deterministic. One
possibility is to allow it’s sub tree to execute, another is to simulate that the sub tree has finished.

The next section will describe what is the protocol controller doing when actually receiving requests and executing it’s handlers. The protocol controller has to update it’s threads whenever a handler method is executed and whenever it finishes.

4.3 Actions Performed upon Method Invocation

When a request arrives, it is placed into the queue of requests. The protocol controller looks into it’s protocol threads’ CanRunTbl[] structures to see whether it can be serviced immediately. If there exists such a thread which can service the request, the protocol controller goes on and erases all the threads which can not service it (CanRunTbl[f_id]==NULL). Then it executes WasRun method on every protocol thread object that is left.

The WasRun method takes three parameters: Node which is a protocol node being executed, Where which is a Context in scope of which was the node executed and Prev which is used to identify the call originator. To get to understand the WasRun method, it is easier to imagine that a message is being sent up the protocol tree from the node Prev to the node Node in scope of the context Where. The Prev node serves to identify, from which side (left or right) did the message arrive.

```cpp
void ProtThread::WasRun(ProtNode *Node, Context *Where,
                        ProtNode *Prev)
{
    if (Node==NULL) return;
    switch (Node->NodeType) {
        case 'F':
            CanNotRun(Node, Where);
            //Removes Node from CanRunTbl.
            WasRun(Node->Up, Where, Node);
            //Propagates the message up.
            break;
        case '+':
            if (Node->Left == Prev) //if call comes from the left
                //we can not execute the right branch any more
                CanNotRun(Node->Right, Where);
            else
                CanNotRun(Node->Left, Where);
            WasRun(Node->Up, Where, Node);
    }
}
```
break;
case ';':
    if (Node->Left == Prev)  // if call comes from the left
        WasRan(Node->Up, Where, Node);
    // Propagate the message up
    break;
case '*':
    if (Where->SeqList->SetRun(Node) )
        // If WasRunBefore attribute of this node status is not set
        // it sets it and propagates the message up.
        // Otherwise it does nothing as this node was executed before
        // and the message was already sent.
        WasRan(Node->Up, Where, Node);
    break;
case '|':
    if (Node->Left == Prev) {  // if call comes from the left
        if (Where->ParallList->SetLeftRun(Node))
            WasRan(Node->Up, Where, Node);
    } else
        if (Where->ParallList->SetRightRun(Node))
            WasRan(Node->Up, Where, Node);
    break;
case '^':
    Context *temp = Where->Next;
    Where->Next = new Context(Where->Parent);
    // Creates a new context with the same Parent as the
    // context Where.
    Where->Next->Next = temp;
    Where->Next->WhatCanRun(Node->Left);
    if ((Where->Parent->RunningCount += 1) == 1)
        // Was ran for the first time
        WasRan(Node->Up, Where->Parent->Parent,
                Node);
    break;
};
};
4.4 Actions Performed upon End of Method Servicing

When a call is finished servicing, the protocol controller executes a Finished method on every protocol thread object. The Finished method updates the Context structure of all the protocol threads. The finished request is removed from the request queue and all sleeping requests that can execute are executed. Thus the event of end of method servicing can result in execution of one or more sleeping requests.

Calling of the Finished method can be viewed as sending a message up the protocol tree telling my parent that I (my sub tree) have finished running.

```c
void ProtThread::Finished(ProtNode *what, Context *where,
                          ProtNode *prev)
{
    if (what==NULL) {
        return;
    }
    switch (what->NodeType) {
        case 'F':
            where->RemoveGuard(what);
            Finished(what->Up, where, what);
            break;
        case '+':
            where->RemoveGuard(what);
            Finished(what->Up, where, what);
            break;
        case ';':
            if (what->Left == prev) {
                // If the call comes from the left
                ActivePT = this;
                where->WhatCanRun(what->Right);
            } else {
                where->RemoveGuard(what);
                Finished(what->Up, where, what);
            }
            break;
        case '*':
            // We need to fork the ProtThread and for one child propagate
            // the Finished call and for the other ask WhatCanRun.
```
ForkedThread = new ProtThread(this_thread);
ForkedThread ->RemoveSeq(what);
    // Remove SeqStatus of this node...
ForkedThread ->RemoveGuard(what);
    // ..., GuardStatus of this node...
ForkedThread->Finished(what->Up, where, what);
        // ... and send Finished to the above node
where->WhatCanRun(what->Left);
break;
case '|':
    if (what->Left == prev) {
        // if call comes from left
        if (!where->ParalList->IsRightRunning(what)) {
            // IsRightRunning will test status of right - if running will set ...
            // ... status of left to finished
            where->RemoveParal(what);
            where->RemoveGuard(what);
            Finished(what->Up, where, what);
        }
    } else
        if (!where->ParalList->IsLeftRunning(what)) {
            // IsLeftRunning will test status of left - if running will set ...
            // ... status of right to finished
            where->RemoveParal(what);
            where->RemoveGuard(what);
            Finished(what->Up, where, what);
        }
    break;
case '^':
    Context::RecurStatus *rs=where->Parent;
r->RemoveContext(where);
if ((rs->RunningCount -= 1) == 0) {
    // Nothing is running any more.
    where = rs->Parent;
    where->RemoveRecur(what);
    where->RemoveGuard(what);
4.5 Management of Guards

Every node in the protocol tree can be guarded. In order to execute a method (which is always a leaf in the protocol tree), all the guards on the path from this method to the protocol root must evaluate to TRUE. The situation is similar to walking through a guarded town. In order to get to the town center, all the guards on the way have to let us pass. Once the guard at the town gate lets us in, we can walk through the town (executing methods) without asking the town gate guard for a permission. Even if the guard's orders change, and he shouldn't let us in, we can still move through the town as we are in already. Once we walk out of the town (finish a protocol sub tree) and want to go in again, we have to get the permission from the guard again.

Let us demonstrate this on an example:

`[guard1]([guard2]method_a)*`

The corresponding protocol tree looks like this:

```
*  
NodeId: 2  
TypeId: 0  
guard1  
   
/method_a  

F  
NodeId: 1  
TypeId: 0  
guard2  
```

The function with `TypeId` equal to 0 is named `method_a`. If we want to execute `method_a` for the first time, both `guard1` and `guard2` must evaluate to 1. But let say we want to execute `method_a` for the second or third time. In these cases, we have to check only `guard2`, as `guard1` already let us in. That is why we have introduced a GuardStatus that tells us weather a guard is active or not.
Generally, before we execute a method we must step up the protocol tree until we find a non active guard (all the guards above the first non-active guard are not active as well). If all the guards on this path evaluate to TRUE, we may set their status to "not active". If one or more of these guards evaluate to FALSE, we can not execute the method.

When a method finishes running, as we propagate the \textit{Finished} message (see section 4.4) we set the guards' status to "active".
5. Protocol Inheritance

One advantage of separating the synchronization code from the code of methods is the possibility to inherit both the synchronization code (the protocol) and the code of methods without experiencing the inheritance anomaly. The enrichable protocols (chapter 2) along with enrichment rules where employed in [PG95] to support inheritance of the synchronization code.

5.1 Enrichment Rule

An enrichment rule has the form

protocol_a \rightarrow protocol_b

The rule means that protocol_a is to be replaced by protocol_b. In order to preserve the "behavior" of the superclass's protocol, protocol_a and protocol_b must satisfy the sound enrichment relation defined in [PG95] as follows:

Let a and b be protocols and tree_a and tree_b are the syntactical trees of a and b in G. We say that b is a sound enrichment of a (a s_enrich b) if tree_b can be constructed from tree_a by a finite repetition of the following 3 actions (enrichment step):

1. Cut a subtree_a (with a nonterminal symbol R in its root) from tree_a.

2. Construct a subst_tree in G containing subtree_a.

3. Modify tree_a by appending subst_tree to the edge which R was cut from; i.e. put subst_tree into the original position of subtree_a in tree_a (the enrichment step).

The above definition of the sound enrichment relation does not, however, provide a way to effectively test for sound enrichment. The following algorithm had to be developed in order to do this. It tests whether Prot_1 \ s_enrich Prot_2 holds for any given protocols Prot_1 and Prot_2 (the input to the algorithm are the corresponding protocol trees Prot_tree1,Prot_tree2).
int Match(ProtNode *N1, ProtNode *N2)
{
    // Returns 1 if protocols with roots N1 and N2 are in
    // the s_enrich relation, otherwise returns 0.
    TSearch *search = NULL;
    ProtNode *N;
    if (!N1) return 1;
    if (!N2) return 0;
    if (N1->NodeType == 'F')
    {
        N = prot_findfirst(&search, 
                    FNCT_BASE+N1->TypeId, N2);
        // finds the first function in a subtree N2 identified by N1->TypeId
    }
    else
    {
        N = prot_findfirst(&search, N1->NodeType, N2);
        // finds the first node in a subtree N2 of type N1->NodeType
        while (N) {
            if (Match(N1->Left, N->Left) &&
                Match(N1->Right, N->Right))
                return 1;
            // Protocols corresponding to the left and right subtrees of N1 are
            // in s_enrich relation with protocols that correspond to the left and right
            // subtrees of T, respectively. If root(Node1) or T are unary operators
            // (*, /), their right subtrees are considered empty.
            else
            {
                N = prot_findnext(&search);
                // finds next node with the same attributes
                // as in prot_findfirst
            }
        }
        return 0;
    }
}

Intuitively, the algorithm looks for a mapping $f$ from Prot_tree1 into Prot_tree2, such that for any pair of corresponding nodes $a$, $b=f(a)$ their properties "being in the left subtree" or "being in the right subtree" are preserved. Note that in case of an unary operation the right subtree is considered empty.

The claim below has to be proven to show that the algorithm complies with the sound enrichment definition. In the claim, we use the following conventions. Let $T$ be a protocol.
tree and let \( U \in T \) be a node of the tree \( T \). By \( U_L \) (resp. \( U_R \)) we denote the left (resp. the right) subtree of the node \( U \), (not including the node \( U \)). Further, \( \sim \) denotes the following equivalence: \( U \sim V \) iff \( U \) and \( V \) represent the same method or the same operator (+, \( \cdot \), |, *, ^).

**Claim (Mapping Claim):**

\[ p \_s\_enrich \ q \]

iff

there exists mapping \( f \) such that:

a) \( f \) is a one-to-one mapping from \( \text{tree}_p \) into \( \text{tree}_q \)

where \( \text{tree}_p \) (resp. \( \text{tree}_q \)) is the protocol tree of \( p \) (resp. \( q \))

b) \( f(V) \sim V \) for every node \( V \in \text{tree}_p \)

c) \( U \in \mathcal{V}_x \Rightarrow f(U) \in f(V)_X \) for every \( U, V \in \text{tree}_p \) and \( X \in \{ L, R \} \)

**Proof:**

Left to right implication:

\[ p \_s\_enrich \ q \Rightarrow \text{there exists a sequence } \text{tree}_p = \text{tree}_1, \text{tree}_2, \ldots, \text{tree}_n = \text{tree}_q \text{ of protocol trees, such that } \text{tree}_{i+1} \text{ was derived from } \text{tree}_i \text{ by one application of the steps (1)-(3); } i \in \{ 0 \ldots n-1 \}. \]

Let us construct the mapping \( f \) from \( \text{tree}_i \) into \( \text{tree}_{i+1} \); which, when nested, will give us the desired relation \( f \), i.e. \( f(N) = f_{i-1} \ldots f_i (f_i (N)) \). The definition of \( f \) is simple; it just maps nodes from \( \text{tree}_i \) onto "itself" in \( \text{tree}_{i+1} \), as by step (2) of \( s\_enrich \) definition, \( \text{subtree}_p \) is contained in \( \text{subst}_\text{tree} \). That is why \( f \) is one-to-one mapping and \( f(V) \sim V \).
For every node \( U \in \text{subtree}_p \) and for every \( V \in \text{tree}_i - \text{subtree}_p, U \in \text{V}_x \Rightarrow f_i(U) \in f_i(V)_x \), as by step (3) of s_enrich definition subst_tree is inserted into the original location of subtree_p. For every other node it is trivially true as the ordering of nodes hasn't changed.

Right to left implication:

If for given protocol trees \( T^i \) of depth \( d \) and \( T^i \) there exists a mapping \( f \) with properties (a)-(c), we can construct \( T^i \) from \( T^i \) by the following algorithm:

\[
\text{for } (i=d-1, i-- , i>=0) \{ \\
\quad \text{for every node } N \text{ of depth } i \{ \\
\quad \quad \text{cut off } N_i \text{ and replace it by } f(N)_i \\
\quad \quad \text{cut off } N_s \text{ and replace it by } f(N)_s \\
\quad \}\}
\]

Every execution of the inner cycle are two valid applications of steps (1)-(3). To show this, we will use induction.

Let \( N^i \) be a node of depth \( d-1 \). If \( N^i \) is not an empty tree, it is composed of exactly one node \( F \), \( f(F) \in f(N^i)_x \), \( f(F) \sim F \), so the subtree with root \( f(N^i) \) can act as subst_tree (step (2) of the definition of s_enrich relation).

Let \( N^i \) be a node of depth \( i \) and \( N^{i-1} \) be it's ancestor. \( N^i_x \) was created by insertion of tree \( f(N^i)_x \), furthermore \( N^i \sim f(N^i) \) and therefore a tree rooted in \( N^i \) is equivalent to a subtree with root in \( f(N^i) \), which is contained in \( f(N^{i-1})_x \). We may perform steps (1)-(3) of the definition, i.e. cut a subtree \( N^{i-1}_x \) and replace it by \( f(N^{i-1})_x \). The idea is illustrated in the figure below.
5.2 Employing Enrichment Rules in Incremental Modification of Protocols

In this section, we present the rule application algorithm as a way of employing enrichment rules for incremental modification of protocols during inheritance.

To illustrate the principle of applying enrichment rules, consider that the protocol \( \text{put; (put + get)} \) defines the synchronization in the class buf. In its subclass xbuf, the synchronization is to be defined by the protocol \( \text{put; (put; x + get)} \). Instead of stating the protocol explicitly in xbuf, we employ protocol inheritance by providing the enrichment rule

\[
\text{Put + get} \rightarrow \text{put ; x + get}
\]

in xbuf. The inheritance mechanism applies the rule on the superclass protocol constructing as the result the protocol \( \text{put; (put; x + get)} \) which will define the synchronization in xbuf.

More formally, let \( p \rightarrow q \) be a rule, \( \text{tree}_p \) resp. \( \text{tree}_q \) be the protocol tree corresponding to \( p \) resp. \( q \), and \( \text{original}_\text{protocol} \) be a protocol. Application of the rule \( p \rightarrow q \) upon \( \text{original}_\text{protocol} \) is defined by the following algorithm (the rule application algorithm):

**Step 1** - Create the protocol tree \( T \) of the \( \text{original}_\text{protocol} \). Mark all its nodes as "unmodified".

**Step 2** - While an "unmodified" occurrence of \( \text{tree}_p \) is found in \( T \), repeat steps 2a, 2b.

**Step 2a** - An occurrence of \( \text{tree}_p \) is found; the corresponding subtree of \( T \) is cut off.

**Step 2b** - Into the emptied space, the right side of the enrichment rule is appended (\( T \) is modified). The appended part of \( T \) is marked as "modified" (to prevent recursive replacements).

**Step 3** - \( T \) defines the \( \text{final}_\text{protocol} \).
The following algorithm performs the protocol enrichment:

```c
void Enrich(ProtNode *OriginalProt, ProtNode *LeftRule,
            ProtNode *RightRule)
{
    if ( OriginalProt == NULL ) return;
    if ( EqualTrees(LeftRule, OriginalProt) )
        // A tree matching the left side of enrichment rule was found
        // in the original protocol.
        // The found tree will be replaced by the right side of enrichment rule.
        ReplaceTree(OriginalProt, RightRule);
    else {
        // Keep looking for the left side of enrichment rule.
        Enrich(LeftRule, OriginalProt->Left);
        Enrich(LeftRule, OriginalProt->Right);
    }
};
```

The left and right sides of the enrichment rule were parsed separately and a protocol tree was generated for both of them. The Enrich algorithm uses pointers to the protocol tree roots as input parameters. It does not generate a separate protocol tree, but modifies the tree corresponding to the original protocol.

5.3 Guards during Protocol Enrichment

Even though, in Sections 5.1 and 5.2 we supposed for simplicity no guards in protocols, there is a simple way of combining guards during protocol inheritance, namely by "ANDing" them. The following two examples illustrate the idea:

original protocol: \([G1]get+anything; [G2]get\)

rule: \( \text{get\rightarrow}[G3](\text{get; xget}) \)

final protocol: \(( [G1\&G3](\text{get; xget})) + anything; ([G2\&G3](\text{get; xget})) \)
original protocol: \([G1]\text{get+anything;} [G2]\text{get}\)

rule: \(\text{get} \rightarrow (\text{get;} [G3]\text{xget})\)

final protocol: \([G1](\text{get;} [G3]\text{xget}) + \text{anything;}\)  
\([G2](\text{get;} [G3]\text{xget})\)

In this example \& denotes the logical "and" and by convention \([G1][G3] = [G1\&G3]\).

The following example illustrates the protocol representation when guards are involved in the process of protocol enrichment:

Let \([G1](a+b) \mid d\) be an original protocol, \(a+b \rightarrow [G2](a+b;c)\) be an enrichment rule. The resulting final protocol is \([G1\&G2](a+b;c) \mid d\).

Original protocol: \([G1](a+b) \mid d\)  
Final protocol: \([G1\&G2](a+b;c) \mid d\)

When tracing the final protocol, all the guards in the list have to return \text{TRUE} in order to allow execution of the protocol subtree.

In this example, the right side of the enrichment rule is guarded by \(G2\), and the occurrence of the left side in the original protocol is guarded by \(G1\), the resulting protocol will be guarded by both \(G1\) and \(G2\), which can be seen as being guarded by one guard \(G1\&G2\). This principle is not implemented by creating another guard \(G3\) equal to \(G1 \& G2\) but by linking the guard we already have (\(G1\) and \(G2\)) in a list pointed by \text{GuardList} (see 2.3). When tracing the final protocol, all the guards in the list have to return \text{TRUE} in order to allow execution of the protocol subtree.

Note that guards are not allowed on the left-hand side of a rule.
5.4 Issues Encountered Relating Protocol Enrichment

5.4.1 Representation Dependent Enrichment

The rule application algorithm as defined in 5.2 may not be practical in all cases, because it is dependent upon implementation of the syntactical tree:

Let \( a+b+c \) be original protocol, \( a+b\rightarrow a+b+d \) be an enrichment rule.

\[
P: \quad \begin{array}{c}
+ \\
\downarrow \\
a & + \\
\downarrow \\
b & \downarrow \\
\downarrow \\
c
\end{array}
\quad \begin{array}{c}
tree_a: \\
\downarrow \\
a & b \\
\downarrow \\
a & \downarrow \\
b & d
\end{array}
\quad \begin{array}{c}
tree_b: \\
\downarrow \\
a & + \\
\downarrow \\
b & \downarrow \\
\downarrow \\
d
\end{array}
\]

As we can see, with the above implementation of syntactical trees, an occurrence of \( \text{tree}_a \) can not be found in \( P \), and therefore the original protocol will not be modified. If the original protocol was written as \( (a+b)+c \), the enrichment rule would take effect. We may also rewrite the enrichment rule to \( a+b+c\rightarrow a+b+d+c \), or \( b\rightarrow b+d \), which would lead to the same final protocol. The “meaning” of these enrichment rules is, however, different than of a rule \( a+b\rightarrow a+b+d \), which may be: “every time we could execute \( a \) and \( b \) in alternative in the original protocol, we may execute it in alternative along with method \( d \) in the final protocol.” To implement this semantics of enrichment rule, more in depth analysis would have to be performed on the original protocol rather than just replacing its subtrees. However, it is not likely that such a semantics can be deterministically defined for every enrichment rule.

5.4.2 Guards in the Original Protocol

Another problem may arise when trying to enrich protocol witch already contains guards. Let \([G1]a+b\) be an original protocol and \(a+b\rightarrow \ldots \) be an enrichment rule.

Original protocol: \([G1]a+b\)

\[
\begin{array}{c}
\leftarrow \quad + \\
\quad \downarrow \\
G1 & \quad a & b
\end{array}
\]

The enrichment steps (2a, 2b) can not be applied as the node \( a \) is guarded by \( G1 \). On the other hand, if \([G1](a+b)\) was an original protocol, it would be enriched to \([G1](\ldots)\).
Original protocol: \([G1] (a+b)\)

In this case the guard is guarding the root (+) of the protocol and there are no other guards "inside" of the protocol tree:

5.4.3 Context Sensitivity

The context sensitivity of enrichment rules has a shortcoming - we need a context in the protocol of a base class to be able to enrich it in a subclass. Introduction of a new method, which does not relate to the methods currently present in the protocol is not possible. We may, however, require the base class to contain a "nucleus construct" allowing for future enrichments; this can be viewed as an analogy with generic classes and abstract methods. For example, the protocol \([^G1]a + [G2]b + \text{dummy}\) can be enriched by \(\text{dummy->dummy} + [G3]c\). Without the dummy method, enrichment would not be possible as guards are not allowed on the left side of a rule.
6. Open Problems, Potential Future Work

The protocol concept is to be viewed more as a paradigm rather than as a specific synchronization mechanism. We didn’t, for example, specify what kind of information guards can access. If guards could access server attributes, the server developer would have to synchronize access to those variables as they could be accessed concurrently by guards and by the server’s methods. This kind of protocol use would totally suppress its benefits.

In [CMH95] it is recommended to strictly separate a synchronization code from a sequential code and synchronization variables from sequential variables. Sequential code is the code of server methods. The programmer of these methods should not need to worry about synchronization of method calls. Sequential code should access only sequential variables. Synchronization code, on the other hand, is used for synchronization purposes only – it should not perform server tasks. It can access separate set of variables – synchronization variables. Protocols with guards fell into the synchronization code category. Logical question arises. Where do we set synchronization variables that we need to test within guards?

In order to design a synchronization mechanism that uses the protocol concept, we need to enable the developer to set synchronization variables. Convenient place for synchronization code is immediately before method execution and immediately after the end of method servicing. Let’s show this on a bounded buffer example:

```java
class buffer {
    synchronized int count;

    void init() prec ode(count=0); 
    char get() prec ode(count--); 
    void put(in char what) post code(count++); 

    protocol(init ; ([count<10]put | [count>0]get)*) ; 
}
```

The class buffer is used to implement buffer of 10 elements. The init method initializes the buffer, get reads one element and put writes one element into the buffer. In the protocol guards, we need to test for the number elements in the buffer – synchronization variable count. The variable is initialized before a call to init - in a prec ode of the init method, it is decremented just before execution of get and incremented immediately after execution of put (in a post code). Note that the synchronization is coded separately from the implementation of methods.

A synchronization mechanism that uses protocols, precodes, postcodes and synchronization variables was implemented in a Students’ Project [MSSV98] as an extension to CORBA/ORBIX [KPT96] [Orbix] architecture.
In order to further increase the expressive power of a synchronization mechanism, we could trigger a synchronization code upon request arrival and allow access of the request queue (Section 3.2). It would enable to perform scheduling tasks within the synchronization code.

Guards should be also able to access invocation parameters of method calls. It would be practical in a case we wanted to implement a method \texttt{Get(n)} that would read \texttt{n} elements from a buffer. The protocol could look like this:

\begin{verbatim}
init : ([count<10]put | [count-n >= 0]get)*
\end{verbatim}

Only guards attached to single methods can access invocation parameters.

As we were implementing several examples, we discovered that it might be useful to make minor modifications to the protocol semantics. At present time, if we want to enable an object method to execute, it has to be explicitly stated in a protocol. The protocol controller would not otherwise allow the method to execute. More convenient interpretation would enable all the methods that are not present in the protocol to execute without synchronization constraints. Thus the protocol would serve to define restrictions to method execution, i.e. synchronization constraints, rather than “permission” to execute as can be viewed in present time.
7. Summary Instead of a Conclusion

In this theses, we have designed the algorithms needed to implement the novel concept of enrichable protocols. A benefit of an enrichable protocol is that it can be inherited and, at the same time, be subject to incremental modification during inheritance. We have discussed the enrichable protocol related issues in the publication [PM97]. Functional algorithms had to be designed in order to show that the concept of enrichable protocols is meaningful.

The protocol is, naturally, coded as a text and therefore has to be transformed into an internal representation. A simple LL(1) syntactical analyzer/parser was developed to produce a protocol tree that is used by a protocol controller.

The protocol controller is the component that is responsible for making decision, whether an arrived request may execute or be put in a queue. The protocol controller implements the algorithm of dynamic protocol tracing to ensure a valid order of executed requests. An important attribute of the protocol tracing is to allow execution of every valid sequence of requests. As there may be many different ways of tracing the protocol, the protocol controller logic always chooses the one that satisfies the most of the requests as they arrive in the FIFO order. During the lifetime of the protocol controller, a table of methods that can execute is being managed. As a request arrives, the protocol controller looks into this table and can immediately decide (in constant time) whether the request may execute or not. The calculations to update the table can be done while the request is executing which significantly reduces delay between the request arrival and its execution.

In order to enable the protocol inheritance, the sound enrichment relation concept was introduced to capture the intuitive notion of sound modification. The sound enrichment ensures that the synchronization policy is "enriched", not turned "upside down" in a subclass. Sound enrichment is based on the idea of enhancing subtrees in the syntactical tree of the given protocol. To make the concept practically useful, we had to develop an algorithm, which tests the sound enrichment relation. The Mapping Claim (Section 5) was proven to show that the developed algorithm complies with the sound enrichment definition.

An enrichment rule is applied through the rule application algorithm. It modifies the original protocol into final protocol by replacing subtrees corresponding to the left hand side of the rule by a subtree corresponding to the right hand side of the rule.

After the above algorithms were implemented, the enrichable protocol concept was tested on several examples. The implementation of algorithms was also used as a base in the Students’ Project [MSSV98]. The protocol concept was found very practical as it greatly simplifies a server object development with respect to synchronization. Few potential enhancements to protocols were listed and tips for implementing a complete synchronization mechanism based on the protocol concept were proposed.
To summarize, the main contributions of this thesis are as follows:

- The design and implementation of the dynamic protocol tracing algorithm used by the protocol controller.
- The design and implementation of the sound enrichment verification algorithm.
- Proof of the Mapping Claim (also published in [PM97]).
- It was a base for the Students’ Project [MSSV98].
8. References


[MWY90] Matsuoka, S., Wakita, K., Yonezawa, A.: Synchronization Constrains with Inheritance: What is not possible - so what is? TR 10, Department of Information Science, University of Tokyo, 1990 (ftp:is.s.u-tokyo.ac.jp)


[Orbix] Iona Technologies Ltd. ORBIX Programming Guide

