Using Knowledge-Based Program Transformation To Develop Radio Software

by

Sahana E. Sarma

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1996

Author .....
Department of Electrical Engineering and Computer Science
May 29, 1996

Certified by .....
John Guttag
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by .....
Thesis Supervisor

Accepted by .....
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Theses

Using Knowledge-Based Program Transformation To Develop Radio Software

by

Sahana E. Sarma

Submitted to the Department of Electrical Engineering and Computer Science
on May 29, 1996, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science and Engineering

Abstract

Efficient development of software is of primary importance to corporations in the electronics industry today. Faulty software costs a company valuable time and money. Because of this need, many companies are investigating methods to speed up software development while maintaining high quality. The goal of my thesis was to evaluate the effectiveness of one such tool. MOUSETRAP uses program transformation to automatically generate code from specifications. I compared and contrasted coding software for a Motorola radio through MOUSETRAP versus manually coding software. Specifically, my thesis project focuses on the design and development of a datalink layer protocol. The development time, the performance, and the code size were all measured between the two versions to draw conclusions about the feasibility of using MOUSETRAP as a software development tool. The results suggest that MOUSETRAP can be an efficient and feasible way to develop commercial software.

Thesis Supervisor: John Guttag
Title: Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Thomas Weigert
Title: Principal Staff Engineer

Acknowledgments

I would like to first thank my advisors Thomas Weigert and John Guttag for helping me improve my thesis through their guidance, valuable time, and effort. I would also like to thank everyone else in the Motorola Research Labs in Schaumburg, Illinois and in Yokohama, Japan who assisted me in my work. Finally, I would like to thank my parents, Venkat and Kamala Sarma, for their constant love and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Software is becoming increasingly important to many products that are a part of our daily lives. Products such as word processors, personal data assistants, and even cellular phones are all primarily controlled by software. Therefore, developing reliable software at an efficient rate is of extreme importance to many companies. In addition, defective software can cost a company valuable time and money. Because of this, there are new tools being developed to improve the software development process. MOUSETRAP is one such tool that is based on program transformation. It is currently capable of automatically generating C code from a set of specifications.

My thesis was an experiment in using MOUSETRAP to generate a piece of commercial software versus generating the same code manually. The goal was to evaluate this process versus the traditional method of software development by developing the same piece of software through both methods and comparing the results. The results were measured by four metrics, development time, code size, performance, and maintainability.

The first part of my project focused on designing the specifications for a datalink layer protocol. The protocol will eventually run on a Motorola radio that runs on a HC16 microprocessor and an operating system called ROS, Radio Operating System. The specifications were designed in an internal Motorola specification language that

is widely used for software development.

The second part of my project required implementing this datalink layer protocol. This involved understanding the OSI, Open Systems Interconnection, model for networks, understanding the role of the datalink layer protocol within that framework, and finally designing and coding the protocol. The coding was done entirely in C.

The third part of the project involved adding a rule scheduler to the MOUSETRAP system. The input to MOUSETRAP is a program in one language, and the output is a program in another language. The program is gradually changed by applying a series of "rewrite rules." The MOUSETRAP system currently has a knowledge base that can transform a program written in an internal Motorola specification language to C. The scheduler tool took as input the rules that are in the knowledge base of the MOUSETRAP system and provided an ordering of these rules which would produce correct code in an efficient manner.

The next part of the project was to identify the special implementation needs of the HC16 and ROS environment. Once, this information was collected, special rules were added to the MOUSETRAP knowledge base to retarget the code to this platform. The specifications were then run though MOUSETRAP to automatically generate the code.

Then, a simulator was written to test the automatically generated code and handwritten code to ensure that each version's behavior was consistent with the specifications. Finally, the feasibility of program transformation as a viable method of software development was evaluated by comparing the generated code to the handwritten code.

1.2 Contributions

This experiment provided empirical data to compare code generated by MOUSETRAP to the handwritten code. This data is valuable to the designers of MOUSETRAP because it demonstrates the benefits and disadvantages of using program transformation as a method of software development and in particular, the benefits and disadvantages of MOUSETRAP. During the course of the experiment, new rules were added to

the knowledge base which will be useful to other MOUSETRAP users. The scheduler tool will help the developers of MOUSETRAP rules to maintain the rule knowledge base. Finally, the complete set of specifications will be helpful to others who wish to code this same protocol.

The transformer has been used before to generate code for other hardware platforms and for other operating systems such as pSOS, and so there are rules in the transformer's rule base that take advantage of these hardware platforms and contain the specialized programming knowledge to produce efficient code in these environments. However, there were no such rules for the HC16 microprocessor and ROS environment. So, while designing the specifications of the datalink layer protocol, I discovered the unique characteristics of this environment to incorporate them into the rule base.

Expanding the rule base with specialized rules for the HC16 using ROS was also useful to other developers who develop code on this platform because they can focus their efforts on the design rather than the special "tricks" of implementing. These "tricks" might influence the way that certain data structures are implemented, or certain rules in the transformer rule base may no longer be valid under this hardware platform. These are ideas that an experienced programmer working with the hardware would know, and incorporating them into rules would be a way to preserve that knowledge.

The scheduler tool produces an efficient order in which the rules should be applied. This is very useful for rule developers because they no longer need to coordinate every time new rules are developed to agree on a new ordering. The scheduler tool simply collects all the rules in the knowledge base and generates the ordering which all of the developers use.

The datalink protocol that was used for the experiment is commonly used in Motorola radios. The only documentation of this protocol is a 10 page, internal Motorola document. This document does not give exact specifications, so there are often questions about implementation details that each engineer chooses to handle in a different manner. Having a clear set of specifications would be helpful to others

within Motorola who wish to implement this particular protocol.

The results of the comparison of the generated code to the handwritten code were especially useful in measuring the practicality of using MOUSETRAP on a widespread basis. The results were measured in terms of development time, code size, performance, and maintainability. The first three factors proved to be approximately the same for both versions, however, it is much easier to maintain specifications than actual code. This proved to be the one of the main advantages of the MOUSETRAP system. If the protocol is modified, then it will be much simpler to change the specifications and then just regenerate the code, rather than modifying actual code. Additionally, much of the complexity of the algorithm can be incorporated very simply into the specifications. This experiment also showed the flexibility of the MOUSETRAP transformer mechanism to be adapted to different environments even though the rules could change depending on the application domain.

The major disadvantage of using MOUSETRAP was ensuring the correctness of the specifications. While I was performing the experiment, there was no method of checking the correctness of the specifications, so the generated code often had to be examined to reveal an error in the specifications. This was tedious and slow.

Overall, this project demonstrated that an automatic code generation system that is based on program transformation is a feasible method of software development in commercial projects. However, it was confirmed that developing a specification validation system is essential to practical use of such a system.

1.3 Related Work

1.3.1 General Program Transformation

The traditional software development process has shortcomings for a number of reasons. One reason is because the specifications of the given problem are not clear enough to reflect the actual function of the code [3]. Furthermore, in the traditional approach, the development is done in one large step to bridge the gap between the

vaguely defined specifications and the precisely defined code [11].

The development of correct specifications is essential to the creation of software. Specifications represent the "contract" between the client and software producer. Additionally, the specification has to be a suitable abstraction of the real-world problem that it represents and is the reference for judging the correctness of the final software product [4].

However, even if the specifications are correct, the implemented software may still not work correctly. This is because the step from the formal specifications to the program is still too large. The idea behind transformational programming is to introduce intermediate steps to this implementation process [11]. This approach was first advocated in 1977 although the ideas were previously presented in 1975 [13]. The major requirements of a program transformation system are support for validating the specifications, performing the program transformations, support for verifying applicability conditions of transformation rules, and tracking the development process [11].

1.3.2 Types of Program Transformation

There are many different technical approaches that are all based on the idea that software development is a stepwise process of applying transformation rules. These approaches generally have different specific goals for the end results of the transformation [11].

The primary goal of most approaches is that of "general support of program modification [11]." This includes

> the development of operational solutions from non-operational problem descriptions, the optimization of control structures, the efficient implementation of data structures, as well as the adaptation of given programs to particular styles of programming (applicative, procedural, machine oriented [11]."

Other goals of program transformation include program synthesis, the automatic

generation of a program from formal specifications, and adapting a program to a particular environment.

The various approaches vary technically along a few categories [11]

- the kinds of languages used for formulating the problem
- the kind of transformation rules used (from syntactic rewrite rules to transformation rules that rely on semantic properties or mathematical theorems)
- different views of the correctness of a transformation rule
- contents and structure of the collection of available rules
- the degree of automation

1.3.3 Program Transformation Systems

There are many transformation systems that have been developed that vary in the ways suggested in the previous section. I will describe just three of the systems that have been developed.

The CIP project, Computer-aided, Intuition-guided Programming, at the Technical University of Munich contributed much to the field of transformation [7]. The CIP system uses the approach of allowing the programmer to guide the transformer system [10]. The knowledge base is small but the fully interactive nature of the system allows a user to build new transformations upon the original transformation rules [5]. The CIP system represents programs as algebraic specifications [12].

Robert Paige's RAPTS (Rutgers Abstract Program Transformation System) is a running transformational program system that does source-to-source transformations on high-level SETL programs. SETL is a high level programming language developed at the Courant Institute of New York that has "syntax and semantics based on the standard set-theoretic dictions of mathematics [5]." RAPTS places many restrictions on the SETL specifications that it can process and relies on a small amount of input from the programmer to guide the transformation process. One unusual feature of RAPTS is that it derives a complexity formula for the specifications that it transforms [9].

Jim Boyle's TAMPR (Transformation-Assisted Multiple Program Realization) system was developed at the Argonne National Laboratory. It mainly deals mainly with modifying Fortran. The TAMPR system is completely automatic. Boyle tests the TAMPR system by comparing code generated from LISP to Fortran to handwritten Fortran. The tables in [2] show the performance of the TAMPR system to be essentially the same as the handwritten code.

1.3.4 The MOUSETRAP System

I will now discuss how MOUSETRAP compares to other program transformation systems. Although the MOUSETRAP system did not have a specification validation system while I was completing my research, one now does exist. The MOUSETRAP underlying framework handles performing the program transformation. Ensuring that the applicability conditions of transformation rules were validated was done manually by creating a rule ordering, but can now be done using a rule scheduler tool. Finally, the development process can be followed by tracing the application of the transformation rules. MOUSETRAP's rule base is organized into "systems," which can be selected by the user, so it is also a fully automated system. Finally, MOUSETRAP uses an internal Motorola specification language for the input programs. This language allows the user to incorporate problem specific information into the specifications as well as into the transformation rules. This allows the system to be powerful in terms of developing efficient code for a particular environment.

Chapter 2

Program Transformation and the MOUSETRAP system

2.1 What is Program Transformation?

Program transformation is a technique of altering a program by the repeated application of "program rewrite rules." These rules state that a given program fragment can be replaced by another program fragment, where these two program fragments do not necessarily belong to the same grammar. The new program should preserve the correctness of the original program.

2.1.1 Uses and Benefits

Program transformation has a variety of applications and benefits. Program transformation could be used for retargeting source code developed for one platform to another platform. At Motorola, base stations generally use an operating system called pSOS while subscriber units use an operating system called ROS (Radio Operating System), or no operating system at all. Code may have been initially developed for either the base station or the subscriber. There is some software that is shared between these otherwise quite different products: the encoding and decoding of data communication and the data protocol software. The shared code therefore has to be

modified to be operational on the other type of hardware with the same overall functionality. This modification generally requires the experience of someone who has the knowledge of ROS, pSOS, or both. Since, this kind of work is frequently necessary, having these rules of modification in a rule base would save this knowledge and might make this process faster in the future.

Program transformation could also be used to produce a more efficient version of a given program. A good example of this is writing code for a parallel machine. Code that is written for a parallel machine is much different than code that is intended to run on a serial machine. Jim Boyle used program transformation to convert pure applicative Lisp to parallel Fortran. In pure applicative Lisp, the order of evaluation of function arguments is irrelevant, and therefore, the arguments to a function could be evaluated in parallel. Boyle utilized this inherent parallelism in Lisp, and created program transformations that would cause as many function evaluations as possible to be actual arguments to lambda expressions. Then further transformations were used to gradually convert this code to parallel Fortran [1].

Program transformation could even be used to automate the generation of executable code from a set of specifications which are in a non-executable language. At Motorola, this type of rule base was developed for converting from an internal Motorola specification language to C. I will refer to this knowledge base as the STOC, specification to C, knowledge base. The specification language is strongly typed, uses reference semantics, and has CLU-style [6] iterators and parameterized datatypes. Deriving code from specifications has several benefits such as standardizing implementations and allowing software engineers to spend additional time on the design phase of a project rather than on the implementation phase. In addition, specifications are often easier to verify for correctness than executable code which results in less errors. Automatic code generation also reduces the labor intensiveness of programming. Finally, specifications are much shorter and easier to understand than the code itself since code becomes longer and more convoluted as the complexity of the underlying algorithm increases.

Program transformation can become even more effective if problem specific knowl-

```
datatype disconnect_pdu
varbits
        CallId::nat 14,
        DisconnectCause::nat 5,
        optional NotifyInd::nat 6 endoptional,
        optional LinkageId::nat 1 endoptional
endvarbits
enddatatype.
```

Figure 2-1: Specification Language Example

edge is incorporated into the system. In communications software, data packet manipulation is extremely important, and it is often tedious for developers to handle the bit manipulation of the data packets. However, by using a language that allows users to express application specific data, this task becomes much easier. Figure 2-1 shows a definition for a disconnect data packet in the Motorola specification language. The notation shows that this packet has two mandatory fields, Caller Id and Disconnect Cause, and two optional fields, Notify Indication and Linkage Id. The notation also indicates the number of bits in each field. The protocol that is used may dictate a particular encoding. For example, the TETRA protocol states that each optional field is preceded by a flag bit to indicate the presence or absence of that field. Program transformation can be used to easily implement these semantics, while the software designer does not have to be aware of the implementation.

The optional bit syntax here provides a simple way for others to understand the structure of the packet. The executable code may lose this quality as the notation become more complicated during implementation.

2.1.2 How does Program Transformation Work?

Program transformation is generally accomplished through the application of a set of rewrite rules which should preserve the correctness of the original program. A rule is applied by traversing the input program, and when there is a match between the left hand side of a rule and a fragment of the input program, the right hand side of the rule replaces that fragment, provided that the applicability conditions of the rules are

met. Program transformation generally proceeds by applying these rules to the given input until no more changes are possible by further application of the rules. If the rule set were confluent, the application order of the rules would not matter because any order of application would result in the same final outcome. One ordering may produce a less efficient program than another ordering, the ordering of the rules may also have an effect on the length of time it takes to transform the program, or the transformation may not even terminate. So, one of the very important issues in using a program transformation system is determining the order in which the rules are applied.

2.2 The MOUSETRAP System

The MOUSETRAP system essentially takes a set of rule sets, each of which contains a few rules, an ordering of these rules, and systematically applies these rules to a given source program. Furthermore, the rule sets are divided into different "systems." A system is a collection of rule sets of that takes the program from one "language layer" to another "language layer." A "language layer" is "characterized by a particular programming model and a collection of notations [1]." For example, one of the systems in the STOC knowledge base is the process system. The process system moves from a language with the *statemachine* construct to a language without this construct.

Each transformation rule in the database has four essential parts:

- an optional variable declaration
- a pattern part
- a replacement part
- an indicator of how the pattern replacement should proceed

```
delay−every is grammar cmod;
?time:Expr;
/. <Expr>'_delay_every(?time)'
==> <Expr>'Delay(?time)'
```

This rule's name is *delay-every* and the grammar is *cmod*, a C grammar. There is only one variable declaration in this rule and that is for the variable *time*. The pattern part is the fragment *_delay_every(?time)*. The variable, *time*, will get matched to the expression that is between the parentheses only if the fragment is of the declared type, *Expr*. The replacement part is *Delay(?time)*. The binding of the variable *time* will be placed in the replacement part.

The replacement indicator is the /. marker. This shows that this rule is a single traversal rule. This mode of traversal ensures that a replacement will occur at most once and that the rule is not reapplied even if there was another pattern match. There are four modes of tree traversal that can be used in addition to single traversal. Repeated traversal ensures that a rule is applied as many times as possible to a given parse tree. Therefore, once a match is located, the same transformation rule is tried again to see if any more matches can be found. There are also traversal modes that do not descend into the tree to find matches, but simply apply the given rule to the current node in the matched tree, only, or apply the given rule to the children of the current node only [15].

The MOUSETRAP system represents the program as well as the pattern of a rule and the replacement pattern as parse trees. When the MOUSETRAP system reaches a particular rule, it compares the parse tree to the rule to see if there is a match. MOUSETRAP provides even more powerful tools to design more complex and convenient transformation rules, as described in section 2.2.2.

2.2.1 STOC Knowledge Base

The STOC knowledge base has many systems. A combination of these systems are currently used for transforming code from the internal specification language to C because an extensive rule base for this purpose has been developed. Figure 2-2 shows the current systems in the STOC knowledge base.

However, the knowledge base could be expanded, for example, to transform specifications to PASCAL if a rule base for this purpose were added.

Currently, the major systems in the STOC knowledge base are the "process" sys-

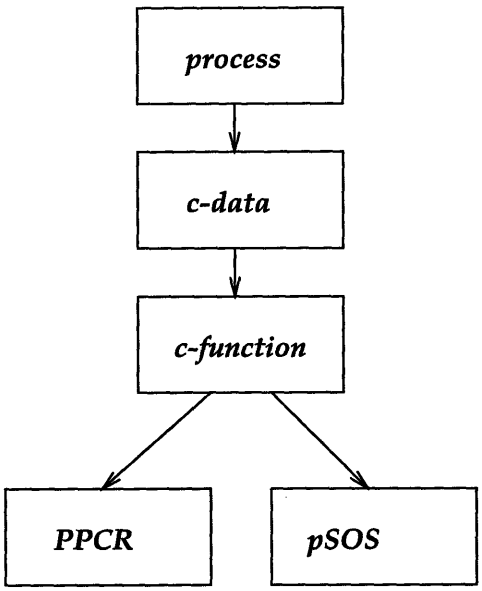

Figure 2-2: Systems of the STOC Knowledge Base

tem, the "c-data" system, the "c-function" system, and finally systems which target the C code to specific operating systems such as pSOS. The systems that target the code to these specific operating systems will be referred to as the "back end" systems. All of these systems work together to transform code that is written in the internal specification language to C.

The process system is the first to be applied. It creates information about each process by analyzing the specifications. It also transforms certain calls, such as calls to send or receive data from a process, to a common interface that all of the back end systems use. The process system also generates tables containing information about the individual processes that the other systems need to access.

The c-data system is the next to be applied. This system analyzes the data specifications and generates information to be stored in tables which the c-function system will need. In addition, the datatypes are converted to C after the c-data system is run.

The c-function system uses the generated tables and then converts all of the functional specifications to C. It is run after the process and c-data system.

Finally, one of the various back end systems is run. There are already back end systems that target the code to a pSOS or UNIX environment. So, the system that

retargets the code for a ROS environment will be another back end system. Once all of these systems are run, the transformation is complete.

2.2.2 Sample Rule Set and Explanation

My primary interaction with the MOUSETRAP system involved increasing the STOC knowledge base by writing rules. The MOUSETRAP system's facilities for writing rules are extremely powerful. Here is one rule that was added to the knowledge base. This rule is converting an *unpack* statement which works on datatypes of figure 2-1 and converts the extraction of fields as bit sequence datatypes to the extraction as record datatypes. The old *unpack* statement is converted to a block of statements. The new statement list is built up by the subtransformations that check how to convert each part of the *unpack* statement. The new statement is then added to the old statement list. Note that MOUSETRAP also has statements such as *for-each* to help write transformation rules.

```
convert−unpack is grammar mtalk;

?var:Expr;
?upl:Unpacklist;
?new,?id:Ident;
?st:Stmt;
?stnew:Stmt;
?el:Exprlist;

global ?stlnew:Stmtlist;                                                    10

/.
<Expr> 'unpack ?var ?upl endunpack'
 & ?stlnew := <Stmtlist> ' '
 &  for each <Unpack> '?id by ?st' in ?upl {    ?el := <Exprlist>'?var'
                                  &  ?stnew:= { ?st /. <Expr> '?id'
                                                    & test* is−accessor−function ?id ?el
                                                                          ==>
                                                                          <Expr> '?id(?var)';
                                              /. <Expr> '?id'                                20
                                                    & ! test* is−accessor−function ?id ?el
                                                                          ==>
                                                                          <Expr> '?id(data(?var))';

                                                  }
                                             & ?stlnew:= <Stmtlist>'?stnew ?stlnew'
```

```
                                                    }
==> <Expr> 'block ?stlnew endblock';
```

As can be seen in this example, every rule set has to be assigned a name and a source grammar. In this case, the name is *convert-grammar* and the grammar is *mtalk*, the name of the Motorola specification language. This rule set also only has one rule. However, if there are multiple rules, there is no precedence between them. At each node of a tree, an attempt is made to apply all of the transformations in a rule set in no stated order. This rule is also a single traversal rule which is indicated by the /. marker.

Chapter 3

The Scheduler

3.1 Why is scheduling transformation rules important?

Currently, the MOUSETRAP system has a very simple control structure. Each rule has a set of conditions that need to be true about the program state before it can be applied, and the application of each rule results in a new set of conditions that describe the state of the program. These conditions govern the ordering of the rules because the rules have to be applied in accordance with their applicability conditions to produce a functionally correct program that reflects the specifications.

The developers of these rules have precisely specified the order of application of the rules to ensure that a correct program is generated. Keeping this ordering updated as new rules are added is essential to the correct operation of the system. However, as many rules have been added by various people it has become increasingly difficult to update the ordering.

One reason that the ordering becomes outdated is the implicit assumptions in the applicability conditions. For example, many rules in the rule base, except for the rule that establishes this condition, depend on the assumption that all of the variables used in the program have been renamed such that each variable has a unique name. This type of global condition is cumbersome to express and is tedious

to check. Therefore, rules which require such global conditions, do not explicitly state them in their applicability conditions. Since there are many rules in the rule base, it is difficult to determine the implicit assumptions of each rule.

There are currently approximately 150 transformation rules in the STOC knowledge base. There is no systematic method of discovering why one rule should be run after another rule. This information is not documented, and only the transformation rule developers have an idea of the dependencies associated with their particular rules. The lack of documentation made it difficult for rule developers to clearly know when a new rule should be applied in the system. This problem often resulted in rules being inserted in the wrong location in the rule ordering and incorrect code was produced by the transformer.

3.2 Goal of the scheduler tool

The scheduler tool's purpose is to provide an automated method for determining the ordering of the rules that preserves the correctness of the program. The tool determines an ordering which takes the least execution time under a set of assumptions. The tool assumes that each rule is entered with the applicability conditions, the post conditions, an assigned language layer, and a cost. The cost is the execution time for a rule and is assumed to be fixed regardless of the specification. It is also assumed that every rule set in a language layer has to be run at least once. So, the ordering is determined at two levels. First, the ordering of language layers is determined, and then the ordering of the rule sets within a language layer is determined.

3.3 Organizing the data

The STOC knowledge base is divided into many systems. There is one section dealing with interprocess control. Another section handles the conversion of the specification language type declarations and global variable declarations to C and is referred to as the data system. Yet another section handles conversion of the functions and

procedures in the specifications to C. Finally, there are several other systems that retarget C code to various operating systems.

The first step was to collect information about the rules. For each rule in the knowledge base, four pieces of information were needed, the applicability conditions for the rule, the resulting post-conditions of the rule, the cost (in execution time of the rule), and a language layer assignment for each rule. The applicability conditions of a rule set represent the conditions that must be established for a program fragment before a rule can be applied. The post conditions describe the new state of the program fragment once the given rule has been run. The cost of the rule is a metric that assigns rules which require longer processing time a higher cost. The language layers are chosen to modularize the design of the rules.

I realized that there were language layers within a particular system in addition to between systems. I will use the c-data system, the system that converts data type declarations from the specification language to C, to describe language layers. The transformation from the specification language to C is not done in one giant step. Instead, constructs are gradually removed or added to the program. After each of these changes, we could consider the program to be in a new "language" with its own syntax and semantics. For example, in this particular module, I concluded that there were essentially three language layers. The first layer is the starting state where all definitions are in the specification language. In this first layer, the data type definitions include record types, union types, enumeration types, type synonyms, constant declarations, global variables, and a special datatype called bit types for designing data packets. However, there is also an intermediate language layer where only three types of specification language definitions exist, record types, union types, and global variables. At this stage, the constant declarations, enumeration types, and type synonyms have been removed. Finally, in the final layer, all of the datatypes are in C. Once the framework for the layers was determined, it was much easier to classify and organize the rules, and assign layers to various rules.

Next, I talked to the rule developers to obtain more information about the functions of each of the rules so that I could describe the applicability conditions and post

conditions for each rule. Finally, I documented the function of each of the rule. Using the collected information, I assigned applicability conditions, resulting conditions, language layers, and costs to each of the rules.

3.4 Algorithm of the scheduler tool

The scheduler tool works in the following manner. It takes in a list of rules which have the pre-conditions (applicability conditions), the post-conditions (resulting conditions), the language layer, and cost. The tool then creates a list of rules for each individual layer, and assumes that every rule in that language layer must be run at least once. Once the rules are classified by layer, the tool finds the ordering of rules at each layer. The overall schedule for a system is then composed of the schedules at each of its language layers. Then using the schedule for each system, the schedule for the rules to convert from the internal specification language to C can be created.

3.5 Finding the optimal scheduling of the rules

I modeled the task of scheduling the rules as a variant of the "optimal path" problem. In this case, the initial point is one set of conditions, and the ending state is another set of conditions. The tool is simply trying to navigate the best route from one set of conditions to another. A route in this case is the schedule of the rules. The tool uses a variation of the A* optimal path search algorithm.

3.5.1 The A* Algorithm

The A* algorithm is a branch-and-bound search, with an estimate of remaining distance, combined with the dynamic programming principle. If the estimate of the remaining distance is a lower-bound on the actual distance, then A* produces optimal solutions [16].

The dynamic programming principle states that "when you look for the best path [from a point] S [to a point] G, you can ignore all paths from S to any intermediate

node, I, other than the minimum-length path from S to I [16]." A branch-and-bound search generally keeps track of all partial paths that are being considered. The shortest path is extended one level, creating as many new partial paths as there are branches. Next, these new paths are considered, along with the remaining old ones. This process is repeated until the goal is reached along some path [16].

So, to conduct an A* search [16],

- Form a one-element queue consisting of a zero-length path that contains only the root node.
- Until the first path in the queue terminates at the goal node or the queue is empty
  - Remove the first path from the queue; create new paths by extending the first path to all the neighbors of the terminal node.
  - Reject all new paths with loops.
  - If two or more paths reach a common node, delete all those paths except the one that reaches the common node with the minimum cost.
  - Sort the entire queue by the sum of the path length and a lower bound estimate of the cost remaining, with least-cost paths in front.
- If the goal node is found, announce success; otherwise, announce failure

3.5.2 Adapting the A* Algorithm to Scheduling

In the case of the scheduling tool, the schedule is found by creating the least cost schedule for each language layer and then combining each of these schedules. Each layer has an initial set of conditions, a set of terminating conditions, and a set of rules in each layer. The optimal path represents the rule ordering to get from the initial set of conditions to the terminating conditions with the least cost, that is, the least computation time.

The A* algorithm is responsible for navigating the path from one set of conditions (the starting point) to another set of conditions (the ending point). Using the information about the layer's initial conditions, the terminating conditions, and the rule database for that layer, an optimal path is determined.

The pre-conditions for a layer are derived from the pre-conditions of the rules assigned to this layer. A pre-condition for a layer is any condition that is an applicability condition for a rule within that layer that is not established by any other rule in that layer. Therefore, that condition must be established by a prior layer.

The terminating conditions for a layer are the pre-conditions of the next layer. Therefore, the terminating conditions for layer n are the pre-conditions for layer (n+1). The A* algorithm then uses the layer's pre-conditions as the starting point and the layer's terminating conditions as the ending point.

The A* algorithm creates new paths by extending the least cost path to all the possible neighbors. Since, I was using the A* algorithm to find a schedule, all of the rules in the database were "possible neighbors" at every step. To reduce the branching factor, I used three guidelines to limit the number of possible neighbors. The first guideline checked that a rule's pre-conditions were valid before it could be included as a neighbor. The second guideline checked that a rule must establish a condition that was not already established to be a neighbor. The final guideline prevented the schedule from traveling down paths that could never be completed by checking that a rule should not remove a condition that would prevent a previously unused rule in the layer from being used, if there is no other rule that could reestablish that condition.

The paths are sorted by the sum of the current cost of the path and an underestimate of the remaining cost. Since, I am making the assumption that every rule in the layer is used at least once, I am able to use A* to determine the least cost path. A path is considered the correct path when it has the lowest sum of current cost and underestimate of the remaining rules. In addition, all of the of the terminating conditions must be established.

3.5.3 Design Issues

There were many assumptions that were made in designing the scheduling tool. One important assumption that was needed to ensure that a schedule would be least cost at a given language layer was that all rules must be used at least once within a layer. This assumption was necessary because, without this assumption, it would be very

difficult to determine which path was least cost because the terminating conditions for a layer could vary.

Another issue was deciding which information users should enter and which information should be derived. The tool needs to be easily used by rule developers so that they will be motivated to enter the rule data in the tool database. Therefore, it was important to limit the information that the user was required to enter. Currently, the user enters a rule set's name, the applicability conditions, the post conditions, the language layer, and a rule's cost. A layer's applicability conditions and terminating conditions are inferred rather than obtained directly from user input.

One problem that exists in the scheduler tool right now is that the applicability and terminating conditions of each rule set has to be entered by the user. This is not only tedious for the user but it is extremely susceptible to errors and can become out-of-date very quickly. The issue here is that all of the developers have to be working out of a common database of applicability and terminating condition names. Another major problem arises when new conditions are needed to describe the applicability or resulting conditions of a rule set, all developers need to understand the exact definition of a given condition so there will be consistency in the description of rule sets.

Another important issue was analyzing the nature of the algorithm to find the optimal path and whether finding an "reasonably good" path rather than an optimal path was sufficient.

3.5.4 Results and Benefits

The tool was able to find a correct path if given the right information about the rules. It quickly provided a schedule for rule application. Using the scheduler tool ensures that there will be documentation for each rule, and that a new schedule will be easily obtainable if new rules are added to the database.

Chapter 4

Background on the Experiment

4.1 Radio Operating System

ROS is the standard operating system for Motorola hand held radios. In ROS, each process is referred to as a task. Each of these tasks has to be defined in an initial configuration file. All tasks in ROS are statically created, and no new tasks can be created dynamically from current tasks which are running.

ROS tasks send information to each other using events, signals, or messages. Events and signals use little memory and are not used for sending data between tasks. Messages are used to send data between tasks. Messages sent between ROS tasks must be predefined and must be included in the initial configuration file so that ROS can determine how much memory is needed.

The memory in ROS is organized by pools and buffers. Each pool has two parameters, the size of the buffers in the pool, and the number of buffers in that pool. So, every time a message is sent between two tasks, the sending task must determine the size of the message, get the pool identifier for that size of message, and finally get a buffer from that pool. The buffer is released by the receiving task once the message is received. Under these conditions, it is extremely important that there are no memory leaks. If there are, then a pool could potentially run out of buffers.

4.2 The Role of the Datalink Layer in the OSI Model

4.2.1 Background on the OSI Model

The model of network layers that is used as a standard is the OSI, Open Systems Interconnection, reference model. The OSI model has seven layers. These layers are the physical layer, the datalink layer, the network layer, the transport layer, the session layer, the presentation layer, and the application layer. The physical layer is layer 1, the datalink layer is layer 2, and so on.

Data transmission between two processes on different machines occurs in the following manner using the OSI model. The sending process has some data that it wants to send to the receiving process. The data is passed to the application layer which attaches the application layer header to the data and passes this new packet to the presentation layer. The process is repeated until the data reaches the physical layer where the data is actually transmitted to the receiving machine.

On the receiving machine, the various headers are stripped off one-by-one as the message propagates up the layers until it arrives at the receiving process. A protocol is the set of rules controlling the format and meaning of the frames, packets, or messages that are exchanged between peer entities (entities in the same layer on different machine) [14].

4.2.2 Function of the Datalink Layer

The datalink layer deals with the algorithms for assuring reliable, efficient communication between two machines. Traditionally, the datalink layer has a number of specific functions. These functions include providing a well defined service interface to the network layer, dealing with transmission errors, regulating the flow of messages so that slow receivers are not swamped by fast senders, and general link management.

The principle function of the datalink layer is to provide services to the network layer. The main service is essentially transferring data from the network layer on the

source machine to the network layer on the destination machine.

The interface between the datalink layer and the network layer uses the standard OSI service primitives. The primitives are request, indication, response, and confirm. Request primitives are used by the network layer to ask the datalink layer to do something, for example establish or release a connection or send a message. Indication primitives are used to indicate to the network layer that an event has happened. Response primitives are used on the receiving side by the network layer to reply to a previous indication. Confirm primitives provide a way for the datalink layer on the requesting side to learn whether the request was successfully carried out.

Another important design issue for the datalink layer is handling flow control, when a sender wants to transmit frames faster than the receiver can accept them. Flow control generally introduces some type of feedback mechanism so that the sender can be made aware of whether or not the receiver is able to keep up. Various flow control schemes are known, but most of them use the same basic idea. The three major schemes are stop and wait, sliding window, and selective repeat [14].

4.3 The Associated Control Procedure

The datalink layer protocol which I am using is called the Associated Control Procedure, also known as the ACP. This protocol uses a sliding window flow control mechanism. This protocol also handles data packets from the network layer which are too large for the datalink layer to send over the communication channel. The datalink layer segments the data packets and reassembles them at the receiving end so that the receiving network layer gets the entire packet in the correct order [8].

4.3.1 The Sliding Window Protocol

The sliding window protocol allows multiple frames to be sent over a channel without waiting for an acknowledgment for each one. In a sliding window protocol, multiple frames are sent to the receiver, and each frame is assigned a number. In the sliding window protocol, one acknowledgment message from the receiver can acknowledge

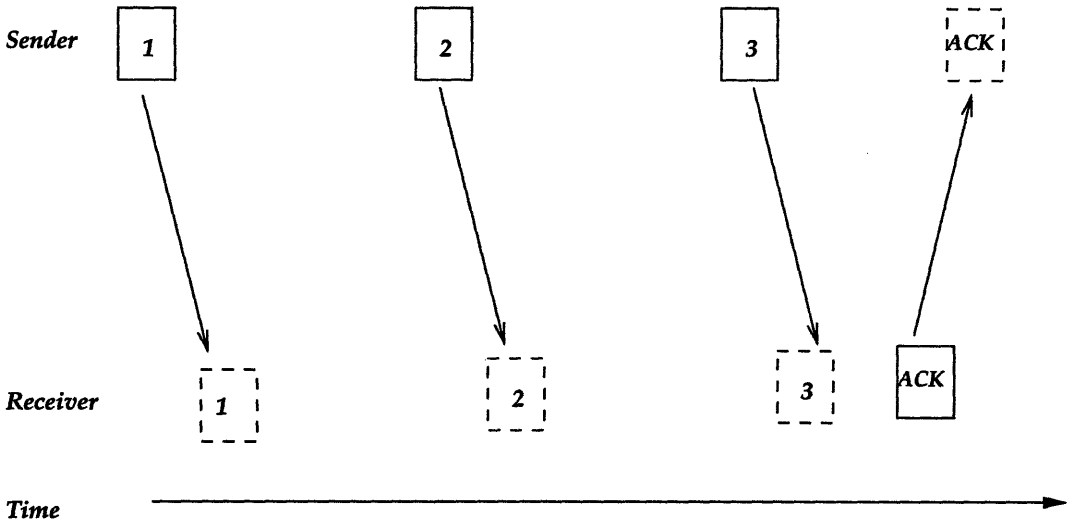

Figure 4-1: The Sliding Window Protocol without Errors

the reception of multiple frames. The receiver in this case only accepts packets which are in sequence and the rest are discarded. This protocol allows for packets to be sent by two parties simultaneously. The sliding window protocol also uses a timer to ensure that the sender will not wait too long before resending a packet if a packet is lost or an acknowledgment is lost. If the timer expires then the oldest frame which has not been acknowledged and all of the frames after that frame are resent [14].

Figure 4-1 shows an example of the sliding window protocol if there are no packets lost during the transmission.

Figure 4-2 shows an example of how the sliding window protocol would behave if packet 2 were lost from the sender to the receiver. Packet 3 would be ignored by the receiver because it would be out of sequence, and packets 2 and 3 would have to be resent.

4.3.2 Overall Hierarchy

Figure 4-3 shows where the ACP fits in with relation to the network layer and physical layer. The ACP is essentially one type of datalink layer protocol. As the figure indicates, the communication is actually occurring vertically with the transfer of data at the physical layer, but the ACP gives the appearance that the communication is

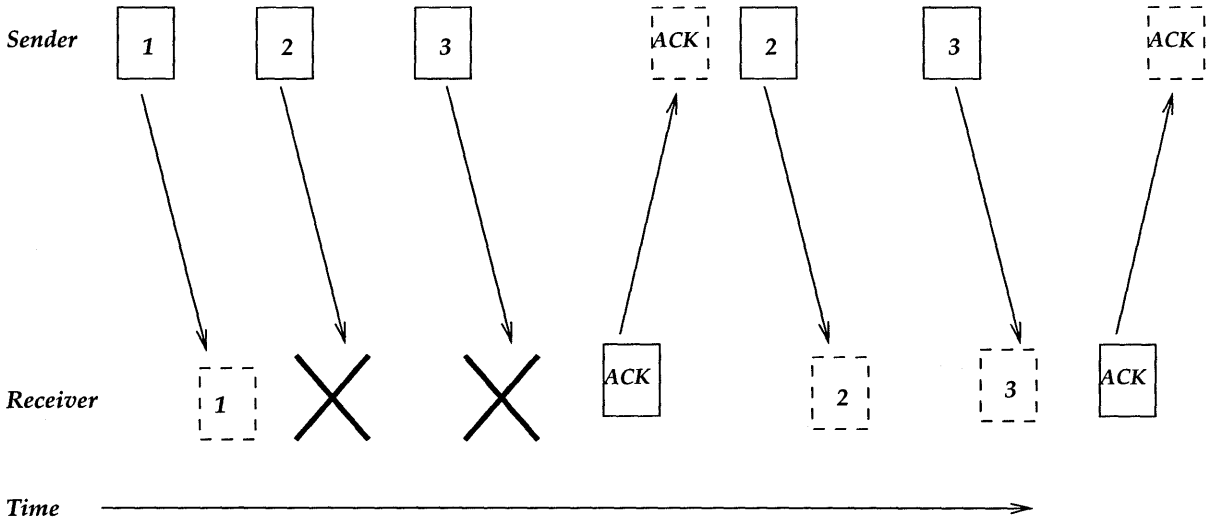

Figure 4-2: The Sliding Window Protocol with Errors

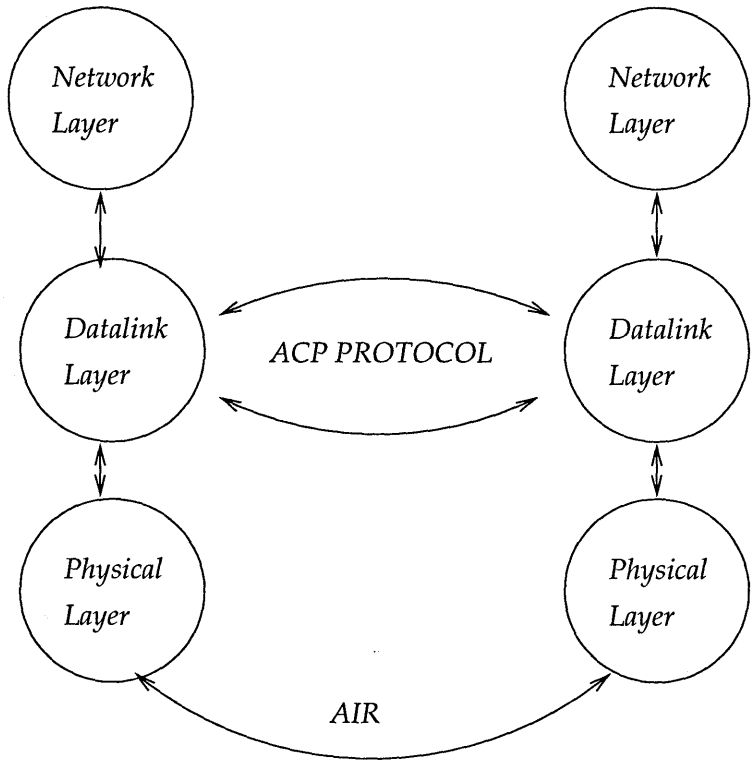

Figure 4-3: ACP Hierarchy

horizontal.

4.3.3 Service Primitives

This section describes the service primitives which the ACP provides to the network layer.

ACP_UNITDATA.request/indication Used for unreliable transfer of data

ACP_ESTABLISH.request/indication/confirm Used for establishment of the reliable transfer mode of operation. There are two types of establishment procedure, normal (no link establishment) and explicit (using primitives to ensure that link exists).

ACP_RELEASE.indication Used to indicate establishment failure

ACP_RELEASE.request/confirm Used for local-end termination of the reliable mode

ACP_DATA.request/indication/confirm Used for reliable transfer of client data while in reliable transfer mode. The data may be any number of bytes.

4.3.4 ACP Structure Diagram

Figure 4-4 shows the general structure of the ACP. There are two channels, Service Access Points, over which the network layer can send and receive data. That is the network layer can be communicating with two other network layer processes simultaneously. The two Peer-to-Peer tasks are identical with one assigned to each channel. The Multiplex task is responsible for routing packets to the correct channel. The Timer task is used by all three tasks and, its only responsibility is to receive "set timer" requests and "stop timer" requests and to return "time out" messages when a given timer expires [8].

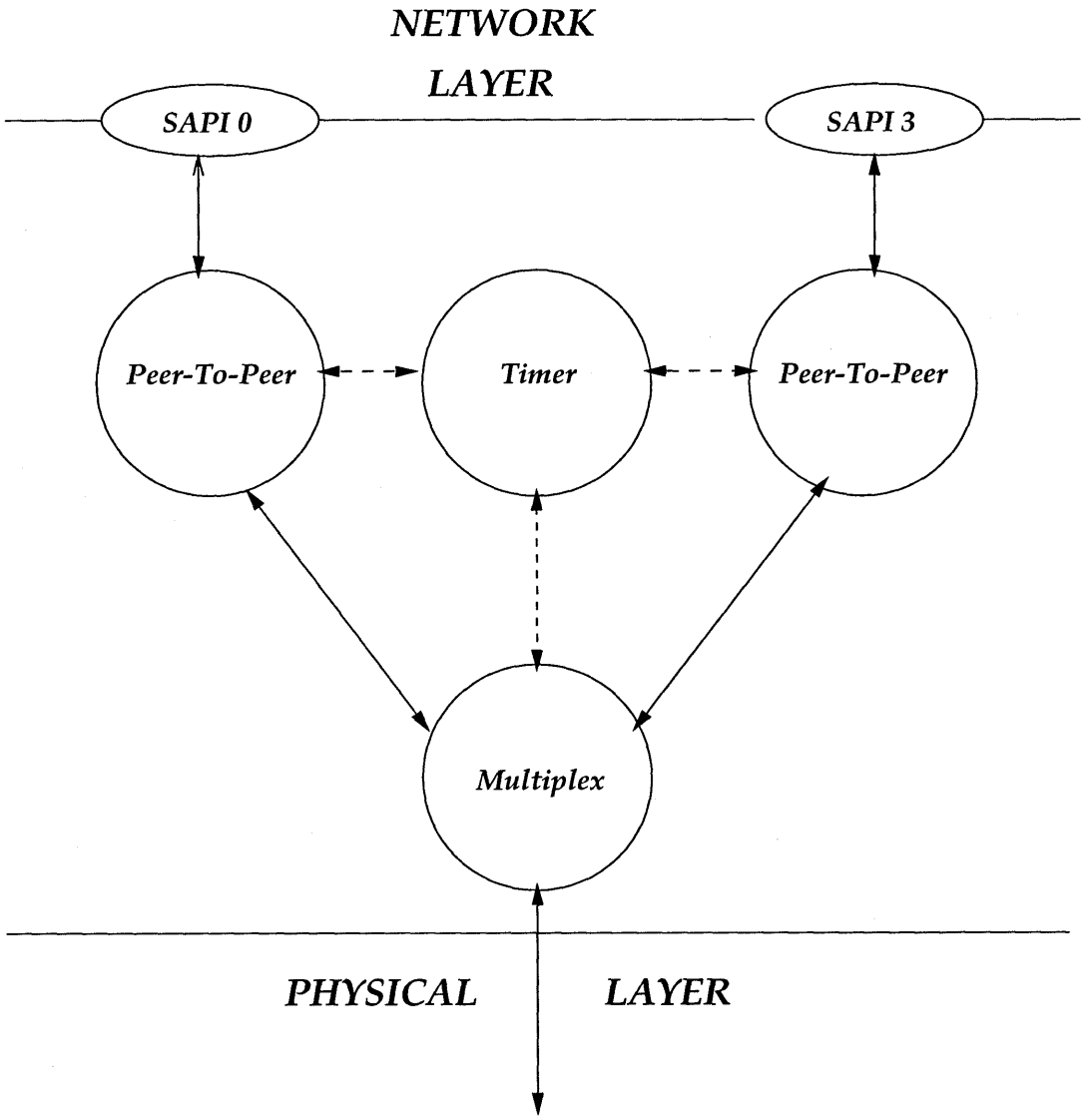

Figure 4-4: ACP Structure

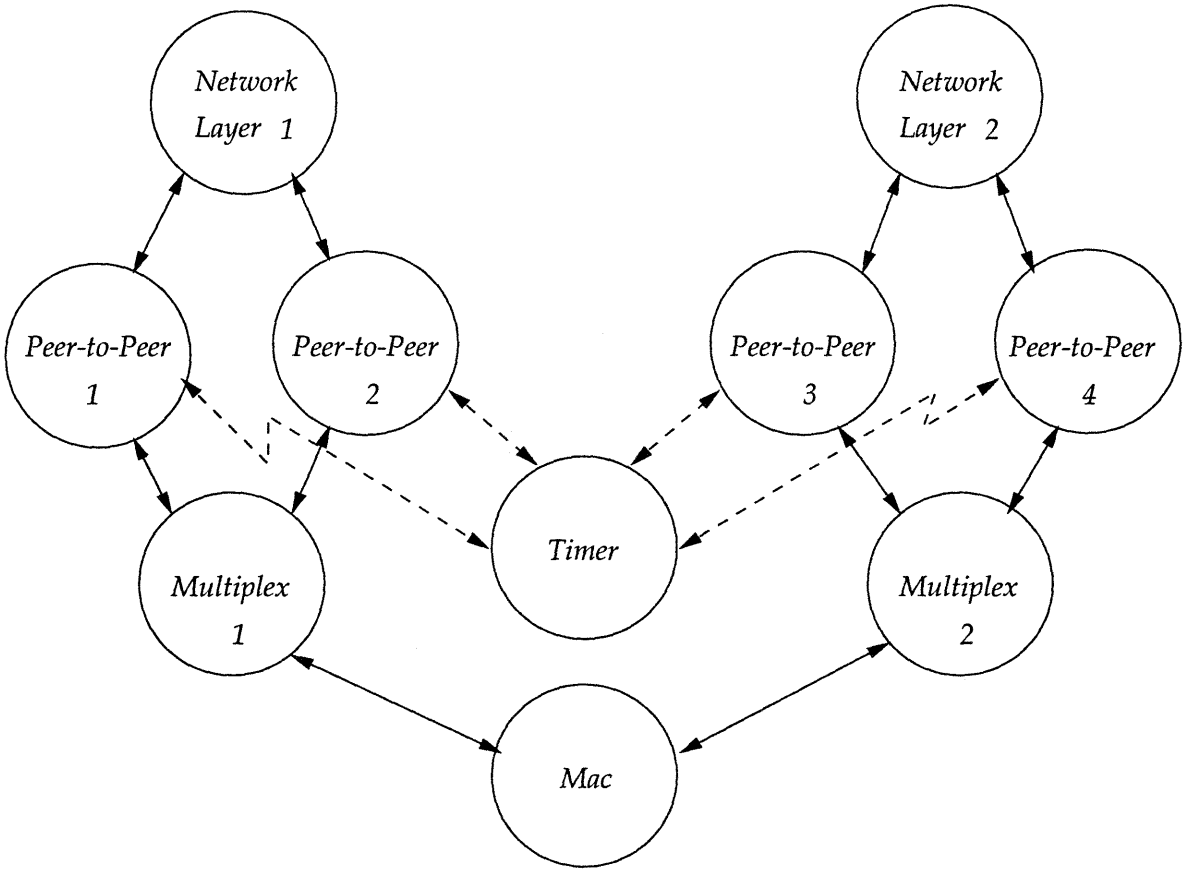

Figure 4-5: ACP Implementation

4.3.5 ACP Implementation

To implement and test the ACP, it was necessary to simulate at least two machines running the ACP protocol and test the communication between them. Each machine then needed a network layer process, two peer-to-peer processes, and a multiplex process. So, for two machines, there were two network layer processes, four peer-to-peer processes, and two multiplex processes. Additionally, one timer process was needed that all of the four peer-to-peer processes would use. Another process was needed to represent the connection to the physical layer, it is generally known as the media access channel, or the MAC process. Figure 4-5 shows the implementation structure of the ACP to simulate communication between two machines. The arrows represent the communication paths between the processes.

Chapter 5

The Development Process

5.1 Traditional Software Engineering Methods

5.1.1 The Method

In industry, the development process traditionally follows a standard pattern. First, a schedule of the development is made, next, specifications are designed for the project. The work is then divided amongst a team of engineers or given to one person who is solely responsible for all of the code. Then, the developers complete their portion of the code. Finally, the code is integrated and tested.

This process seems logical, yet often times there are many problems that develop during the development cycle. One problem that commonly occurs is the specifications are changed during the course of development. Design changes can often require the modification of many engineers' code. The time needed to adjust the code will vary from person to person and can seriously delay the schedule of a project.

Another problem that often delays projects occurs when developers are depending on another person's knowledge about a particular subject matter. An example of this often occurs when developers are trying to port code to a particular platform. The details of that platform need to be known, and the person responsible for porting the code needs to depend on the platform "expert."

5.1.2 The Experiment

The process of developing the ACP code was divided into three major steps. First, I had to learn about the application domain. This required understanding how data protocols work, the details of the datalink layer, and the details of the ACP. Next, the specifications were designed. The specifications were developed in the internal specification language. Finally, the specifications needed to be implemented in C. I was the only engineer working on the actual coding, so there was no division of the work. The code was then tested.

5.2 Program Transformation

5.2.1 The Method

The process when using program transformation as the method of software development varies from the traditional approach only in the middle stages. When using program transformation for development the major change occurs in the way the coding is done. Rather than coding the specifications by hand, the code is generated by the transformer. The steps in automatically generating code are developing and designing the specifications, adding rules to the knowledge base if necessary, and finally running the specifications through the transformation system.

The problems discussed above which are often associated with development are no longer an issue when using program transformation. First, as soon as the specifications are changed, the code is simply processed by the transformer again, and the generated code is tested. So, there are no issues about delays occurring because of modifying code. Delays occurring because of waiting on an "expert" are hopefully also eliminated because the domain specific knowledge should be embedded in the knowledge base. Moreover, if the necessary rules are not yet in the knowledge base, then the "relevant knowledge" has to be collected only once. One example of this that I will discuss further in the next chapter, deals with using a timer function using ROS. It is not readily available in ROS, but it is a feature that almost all protocols

require. Therefore, I had to add a timer feature that any developer who needed a timer would be able to use.

5.2.2 The Experiment

So, when using the MOUSETRAP system, first, specifications for the ACP were developed in an internal Motorola specification language. I then added rules to the MOUSETRAP rule base to ensure that the specifications would get implemented correctly. Specifically, I had to add a set of rules that would target the code for the ROS operating system. Then, the code was tested on the same test cases as the handwritten code to ensure that the behavior was correct.

Chapter 6

Adding New Rules to the Knowledge Base – The ROS System

This chapter describes the role of the new rules that were developed to target code to ROS, and the function of some of the essential rules that were added.

6.1 Purpose of the New Rules

The following rules were added to the STOC knowledge base to allow C code to be targeted for ROS. Figure 6-1 shows how the ROS system fits into the STOC knowledge base. Currently, the C code generated by MOUSETRAP can be targeted to three operating systems, pSOS, PPCR (see section A.3), and ROS. I will refer to these three systems as the back end systems.

A typical ROS message has as a header and then a series of data fields. Since, different messages require different pieces of data, there are different message sizes and therefore different pools. To avoid needing multiple pools because of varying message sizes, a standard ROS message definition was designed so that the code could be standardized. Using the standard definition, only one ROS pool is needed. The rules below assume that a ROS message has the definition in figure 6-2.

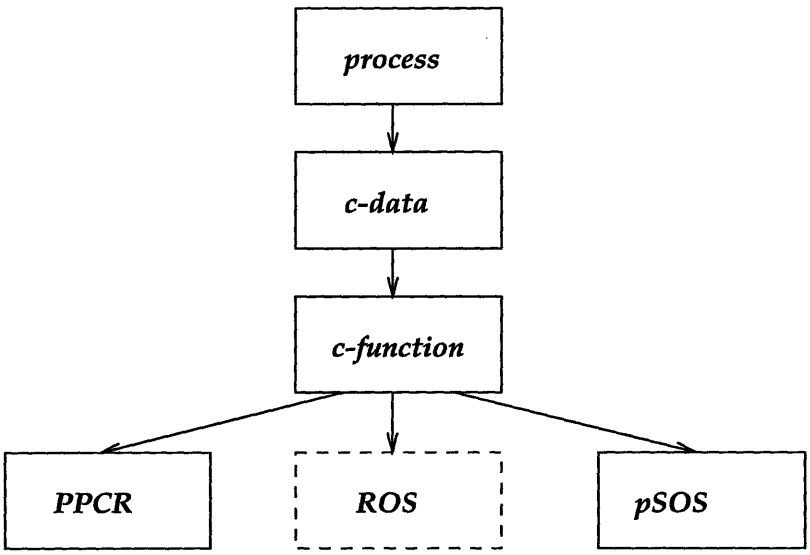

Figure 6-1: Role of the ROS system in the STOC Knowledge Base

```
typedef struct ros_message
{
  BUF_HEADER ros_header;
  unsigned int signal;
  unsigned int instance;
  union {
    void *pointer;
    struct _bsqrep bitseq;
    }value;
} Ros_message;                                                    10
```

Figure 6-2: ROS Message Definition

The reason the ROS message has been organized in this way is because it follows a convention that the process system establishes. The process system assumes that every message that is sent between processes has three parts regardless of the eventual operating system, a signal field, a instance field, and a value field.

The signal field represents a message's signal value. The instance field in ROS is always set to zero because there is only one instance of every task, but is necessary because the process system introduces this as a possible value. The value field is either a structure which represents a bit sequence or a pointer to another data structure.

Each of the back end systems has a message structure with atleast these three fields. There are additional fields if it is necessary for that particular operating system. An example of an additional field is the BUF_HEADER field in figure 6-2. This field is required in all ROS messages.

6.2 The ROS Knowledge Base

Approximately 10 rule sets were added to the STOC knowledge base to allow C code to be targeted to ROS. The ROS system assumes that the process system, the c-data system, and the c-function system have already been run. The next section will focus on some of the more significant rule sets in the ROS system.

6.2.1 Sending Messages in ROS

The process system converts all requests to send messages to a *OS_SEND_MESSAGE* command. This rule translates the *OS_SEND_MESSAGE* command to the appropriate code for ROS. The rule calculates the size of the message, gets the pool identifier for this size of message, gets a buffer from this pool, and finally sends the message to the appropriate task. Each of the back end systems has a rule which converts the *OS_SEND_MESSAGE* statement to the appropriate code to send a message in that operating system.

```
send−C−expand is grammar cmod;

?taskid : Ident;
?newtaskid : Ident;
?signal : Expr;
?value : Expr;
?instance : Expr;
?newchar : Charconst;
?procname:Ident;
                                                                    10
/.
  <Stmt>'OS_SEND_MESSAGE(?procname,?signal,?value, ?instance);'
        & ?newtaskid := eval string−upcase ?procname
    ==>
  <Stmt>'{
```

```
Ros_message * send_buf;
POOL_ID send_pool_id;
send_pool_id = Get_Pool_Id((sizeof(Ros_message) - sizeof(BUF_HEADER)));
send_buf = (Ros_message *) Get_Buf(send_pool_id);

send_buf->signal = ?signal;
send_buf->value = ?value;
send_buf->instance = ?instance;

Put_Msg((BUF_PTR) send_buf,Get_Task_Id(?newtaskid));
  }' (typedef POOL_ID, Ros_message, BUF_HEADER,BUF_PTR)
```

6.2.2 Receiving Messages in ROS

The process system introduces the *_get_input_queue()* statement. This statement indicates that a process is going to wait to receive a message. This rule handles the conversion of the *_get_input_queue()* statement to appropriate ROS code. Again, each of the back end systems has a rule to handle this conversion. Notice, that the *Free_Buf* statement is automatically inserted to ensure that there will be no memory leaks. Forgetting to free memory is a common programming mistake which can often be avoided using program transformation.

```
ros-get-input-queue is grammar cmod;
?ul : Unitlist;
?typeid : Type;
?fname : Ident;
?parmlist : Parameterlist;
?decllist : Decllist;
?sl : Stmtlist;
?m : Ident;
?var : Ident;

/. <Func>'
?typeid  ?fname ( ?parmlist )
{ ?decllist
  ?sl
}'
  & <Stmt>'_get_input_queue();' <= ?sl
  & ?m  := new <Ident>'m' #since we have a _get intro a msg var
  & ?sl := ?sl /. <Stmt>'_get_input_queue();'
            ==>
              <Stmt>'{ ?m = (Ros_message *) Get_Msg();}' (typedef Ros_message);
                          # expand _get_signal_name()
               /. <Expr>'_get_signal_name()'
                    ==>
```

```
                <Expr>'_get_signal_name (?m)'
                ;
                        # expand _get_instance_number();
              /. <Expr>'_get_instance_number()'
                  ==>
                 <Expr>'_get_instance_number (?m)'
                ;                                                          30
                        # expand get_value
              /. <Stmt>'?var = _get_value(?var);'
                  ==>
                        <Stmt>'{?var = _get_value(?m,?var);
                                Free_Buf((BUF_PTR) ?m);}' (typedef BUF_PTR)
                                                        ;
                    /. <Stmt>'delete_thingy ();'
                        ==>
                      <Stmt>'{
                                delete_thingy (?m);                          40
                                Free_Buf((BUF_PTR) ?m);}'(typedef BUF_PTR)
                        ;
==> <Func>
'?typeid ?fname ( ?parmlist )
 { ?decllist
   Ros_message * ?m;
   ?sl
 }' (typedef Ros_message)
;
```

6.2.3 Getting Message Data in ROS

There are three main parts to a ROS message, the signal, the process instance and the actual data being passed. There are three statements which correspond to these parts of the message and are used in conjunction with the *_get_input_queue()* statement. These are *_get_signal_name()*, *_get_instance_number()*, and *_get_value()*. This rule converts each of these statements into the correct ROS code. Once again, each of the back end systems has a corresponding rule that performs a similar function.

```
ros-get-slots is grammar cmod;

?var, ?m: Ident;
?e : Expr;

/.  <Expr>'_get_signal_name (?m)'
==> <Expr>'?m -> signal' ;

/.  <Expr>'_get_instance_number (?m)'
==> <Expr>'?m -> instance' ;                                               10
```

```
/.  <Stmt>‘?var = _get_value (?m, ?e);‘
==> <Stmt>‘?var = ?m -> value ;‘ ;
```

6.2.4 ROS Timer rules

The ACP required the use of a timer. This is a typical need of almost all protocols. Unfortunately, ROS did not have a timer feature readily available. To create a timer that could be easily used by anyone writing specifications, I developed a ROS timer task that would run periodically. The timer task simply required that the user specify the number of timers needed for the application. A user writing specifications could use *start_timer* and *stop_timer* calls in the specifications. The *start_timer* call required three arguments, a timer name, the length of the timer, and finally the process that should be notified when the timer expired. This timer would get added to the list of active timers. The *stop_timer* call required two arguments, the timer name and the process that was scheduled to receive the notice when the timer expired. If the timer specified by *stop_timer* was not in the list of active timers, then no action occurred. Otherwise, the specified timer was stopped.

The period of the timer task was considered a clock tick. At every clock tick, each of the timers that were started would be decremented. If a timer expired, then a “time out” message was sent to the appropriate task. This simple timer is usable on any platform, regardless of hardware timing, and allows multiple timers to coexist while only having one timer task running.

Chapter 7

Results

This chapter focuses on the comparison between the code that was developed through traditional software engineering techniques, and the code that was developed through program transformation and MOUSETRAP. When comparing the code, I examined four different categories, the time for development, the size of the code, the performance, and the maintainability of the code.

7.1 Development Time

The major steps in developing the code through traditional software development were learning about the application domain, writing the specifications, and coding the protocol. Learning about the application domain took approximately a month. Writing the specifications also took approximately a month, and the actual process of coding the protocol took approximately one month. The total time for the entire development was then about three months.

Normally, the steps for developing the code through program transformation are learning about the application domain, writing the specifications, writing the rules if necessary, and finally running the specifications through a transformer. In the case of this experiment, I only needed to add more detail to the specifications, write the rules to target the C code to ROS, and run the specifications through MOUSETRAP.

The time to adjust the specifications that were originally developed was approxi-

mately one month. The specifications had to be adjusted because specifications that are used in the traditional coding process do not need to be as detailed as the specifications that the transformer requires. Developing the rules for the ROS environment took approximately a month as well. Finally, I had to run the specifications through MOUSETRAP. The system took approximately two days to generate all of the code. Note that if the development had been done solely through program transformation then the time needed to learn about the application domain would also be added to the development time.

So, the same code took approximately two months to develop using MOUSETRAP and program transformation. However, if code needed to be developed for the ROS environment again, the time to develop the rules would not be necessary. Note further, that specifications and additional rules can be developed concurrently and would probably not be developed by the same personnel. Thus, it could take approximately 1 calendar month to develop this application using MOUSETRAP.

7.2 Source Code Size

7.2.1 Data

Table 7.1 allows a comparison between the specifications, the code that was generated by MOUSETRAP, and the code that was written manually. Each of the major processes is listed with the corresponding sizes for these three categories. Note that there are four "peer-to-peer" processes and two "multiplex" processes. Note that the expansion factor from the specifications to the generated code varies depending on the procedure. The size of the procedure *segment* essentially doubled while the size of the *initiate_retransmission* stayed essentially the same. The specification language is similar to C, but it can be used to encapsulate an idea that is sometimes complicated to generate.

Table 7.2 provides a similar breakdown of the code size of the main procedures of the ACP. Again, the code size is shown for the specifications, the code that was

Process Name	*Specification (Bytes)*	*MOUSETRAP (Bytes)*	*Traditional (Bytes)*
network layer	1762	3576	2853
peer-to-peer	6054	38574	6696
multiplex	3166	18199	4210
mac	1517	4129	2409

Table 7.1: Breakdown of Process Source Code Size

generated by MOUSETRAP, and the code that was written manually.

Combing the data from table 7.1 and table 7.2, the size of the rest of the code, and the size of the datatype declaration, the overall size of the code can be determined. Table 7.3 shows this calculation.

Tables 7.1, 7.2, and 7.3 all measure the source code size strictly by bytes. Another metric that is used to measure the size of C source code more accurately is the number of semicolons in the code. This number gives the approximate number of instructions in the source code. Table 7.4 shows the number of instructions in each of the processes, and table 7.5 shows the number of instructions in the major procedures of the ACP. These tables simply compare the handwritten code and the MOUSETRAP generated code. Table 7.6 shows the overall number of instructions in both versions.

7.2.2 Interpretation

Table 7.3 shows that the code generated by MOUSETRAP was approximately three times the size of the specifications, while the code that was developed manually was approximately the same size as the specifications. Table 7.6 shows that the code generated by MOUSETRAP was approximately six times the size of the specifications. This data clearly demonstrates that there is a large difference between the source code size of the generated code and the handwritten code. The difference between the code sizes can be attributed to a number of reasons.

Procedure Name	*Specification (Bytes)*	*MOUSETRAP (Bytes)*	*Traditional (Byte*
segment	4324	9143	2462
advance_upper_window	611	2212	521
advance_lower_window	1401	2863	1263
send_pdu	2731	8540	3317
initiate_retransmission	248	362	326
distribute_pdu	860	1446	581
advance_receive_window	1655	7996	1397
reassemble	988	2098	625

Table 7.2: Breakdown of Procedure Source Code Size

Piece	*Specification (Bytes)*	*MOUSETRAP (Bytes)*	*Traditional (Bytes)*
code	84768	307039	93344
datatype declarations	8042	2519	4882
overall code size	92810	309558	98226

Table 7.3: Calculation of Overall Source Code Size

Process Name	*MOUSETRAP (Semicolons)*	*Traditional (Semicolons)*
network layer	100	54
peer-to-peer	1094	153
multiplex	465	87
mac	115	53

Table 7.4: Breakdown of Process Instruction Size

Procedure Name	*MOUSETRAP (Semicolons)*	*Traditional (Semicolons)*
segment	276	51
advance_upper_window	57	8
advance_lower_window	75	19
send_pdu	231	47
initiate_retransmission	7	5
distribute_pdu	39	8
advance_receive_window	224	15
reassemble	61	10

Table 7.5: Breakdown of Procedure Instruction Size

Piece	*MOUSETRAP (Semicolons)*	*Traditional (Semicolons)*
code	6591	1110
datatype declarations	74	116
overall code size	7891	1226

Table 7.6: Calculation of Overall Instruction Size

First, MOUSETRAP inlines many functions that deal with basic types such as queues and linked lists into the actual code and so there are many sections of repeated code. In this case, all of the code to send and receive messages in ROS was inlined. A programmer would traditionally put these pieces of repeated code into a function rather than inlining the code. Notice that in the generated code for the procedure *Reassemble*(see section 7.6.3), the procedures *get_bytes_length* and *put* from the manually generated code(see section 7.6.2 were inlined. Additionally, since MOUSETRAP is not a compiler, it generates many temporary variables through the transformation rules which are superfluous and add to the code size.

Finally, the size of the code in the specification language is very similar to the source code size of the handwritten code. This is because the specification language is very similar to C, and therefore, it is relatively easy for a programmer to correlate the specifications to C.

It is interesting to note, however, that the size of the datatype declarations was the biggest using the specification language, and was the smallest when generated by MOUSETRAP. This is because many of the datatypes were stored in tables and just inlined into the code rather than defined as a type. One example of this was the enumeration types that were declared in the datatypes specification. These enumerations were stored in a table and automatically inserted into the generated code, making the type declarations unnecessary.

7.3 Object Code Size

7.4 Data

There is a significant difference between the source code size of the MOUSETRAP generated code and the handwritten code size. However, the code generated is intended to run on hardware. Therefore, the comparison of the object code size between the generated code and the handwritten code is extremely important.

Object code size determines how much memory a particular piece of hardware will

Process Name	*MOUSETRAP (Bytes)*	*Traditional (Bytes)*
network layer	3934	3934
peer-to-peer	9402	6710
multiplex	6686	4157
mac	4464	3891

Table 7.7: Breakdown of Process Object Code Size

require, and additional memory adds to the cost of a particular product. Therefore, software developers are constantly trying to reduce the size of object code. Table 7.7 shows the object code sizes of the processes, table 7.8 shows the object code sizes of the major procedures, and table 7.9 shows the overall object code size.

7.4.1 Interpretation

The data in tables 7.7, 7.8, and 7.9 show that the object code produced is approximately twenty KBytes bigger. Although the source code generated by MOUSETRAP was significantly larger than the handwritten code, the object code size of the two versions are not that much different.

7.5 Performance

Three tests were used to compare the performance of the generated version and the handwritten version. These tests were the establishment of a connection, sending a data packet under reliable mode, and sending a data packet under unreliable mode. Each of these tests were also performed at different error rates in the physical layer transmission. The performance was tested with a 0% error rate, a 16% error rate, and a 20% error rate.

Procedure Name	*MOUSETRAP (Bytes)*	*Traditional (Bytes)*
segment	2950	1071
advance_upper_window	865	645
advance_lower_window	1407	941
send_pdu	2778	2459
initiate_retransmission	482	707
distribute_pdu	1079	676
advance_receive_window	1981	1000
reassemble	1205	764

Table 7.8: Breakdown of Procedure Object Code Size

Piece	*MOUSETRAP (Bytes)*	*Traditional (Bytes)*
process	63312	46913
procedures	12747	8263
overall code size	76059	55176

Table 7.9: Calculation of Overall Object Code Size

Error Rate(Percent)	*MOUSETRAP (Seconds)*	*Traditional (Seconds)*
0	10	11
16	14	16
20	24	22

Table 7.10: Time to Establish a Connection

7.5.1 Establishing a Connection

The first test is the establishment of a connection between the *nl1* process, the network layer 1 process, and the *nl2* process, the network layer 2 process (see figure 4-5). The establishment of a connection if *nl1* wants to initiate a connection requires that *nl1* first sends an establish request for a particular channel to its respective *peer-to-peer* process. Then, the *nl2* process receives an establish indication message. Finally the *nl1* process receives an establish confirm message. The times that are shown in table 7.10 show the time it took in seconds from the *nl1* process to send the establish request message until the establish confirm message was received.

7.5.2 Sending Reliable Data

The second test was a reliable data request between the *nl1* process, the network layer 1 process, and the *nl2* process, the network layer 2 process (see figure 4-5). The *nl1* process is trying to send a 256 byte message to the *nl2* process. Sending a reliable request requires that *nl1* first sends a reliable data request to the channel where a connection has been opened. Then, the *nl2* process receives an data indication message when the data has been received. Finally, the *nl1* process receives an reliable data confirm message to ensure that the *nl2* process has received the data. The times that are shown in table 7.11 show the time it took in seconds from the *nl1* process to send the reliable data request message until the reliable data confirm message was

Error Rate(Percent)	*MOUSETRAP (Seconds)*	*Traditional (Seconds)*
0	57	54
16	170	176
20	347	340

Table 7.11: Time to Send a Reliable Data Message of 256 Bytes

received.

7.5.3 Sending Unreliable Data

The final test was a unreliable data request between the *nl1* process, the network layer 1 process, and the *nl2* process, the network layer 2 process (see figure 4-5). The *nl1* process is trying to send a 256 byte message to the *nl2* process. Sending a unreliable request requires that *nl1* first sends an unreliable data to either channel. Then, the *nl2* process receives an unreliable data indication message when the data has been received. In this case, no confirmation message is sent to the *nl1* process. The times that are shown in table 7.12 show the time it took in seconds from the *nl1* process to send the unreliable data request message until the *nl2* process received the unit data indication message.

7.5.4 Performance Summary

It can be seen from table 7.10, table 7.11, and table 7.12 that the performance is essentially the same between the two versions. The minor variations here between the times are not significant. However, it can be seen that the error rate does have a significant effect on the time of the transmission.

Error Rate(Percent)	*MOUSETRAP (Seconds)*	*Traditional (Seconds)*
0	2	2
16	2	2
20	dropped	dropped

Table 7.12: Time to Send a Unreliable Data Message of 256 Bytes

7.6 Maintainability

Maintainability can in general be divided into two categories, readability and extensibility. By readability, I refer to the situation where someone who is not directly associated with the development of the code is trying to look at the code and understand the procedures. By extensibility, I am referring to the ability of other developers to expand the features that the code implements. It is important to see an example of the specifications, the generated code, and the manually written code to discuss the maintainability. The following example is of the Reassemble procedure from the ACP.

7.6.1 Specification

```
procedure Reassemble
  (rx_pdu::own ACP_PDU,processdata::ProcessVars_Type)
  locals
    tx_pdu::own ACP_PDU:=new ACP_PDU,
        bytes_length::nat:=0,
        tx_data::own ByteSequence_Type:=new ByteSequence_Type,
        tx_data2::own ByteSequence_Type:=new ByteSequence_Type,
        l3_length::L3Length_Type,
        length_val::nat,
        seq_length::nat                                          10
  endlocals

  for each elem::IPDUQueue_Type in _queue$elements_of(InSequenceQ(processdata)) do
```

```
      length_val:=acptob(length(elem)).
         bytes_length := bytes_length + length_val.
   endfor.

   l3_length := btol3(bytes_length).
   seq_length:= l3tob(l3_length).
   tx_data2:=build_sequence(init_sequence(extract(tx_data),l3_length,seq_length),l3_length,processdata).
   tx_pdu:=build_data_indication_pdu(sapi(rx_pdu),extract(tx_data2),l3_length).

   case NetworkLayerId(processdata)
         is 'nl1 then put(via(L31),ACP_DATA_indication(tx_pdu),to()).
          elseis 'nl2 then put(via(L32),ACP_DATA_indication(tx_pdu),to()).
   endcase.
endprocedure.
```

7.6.2 Manually Generated Code

```
void Reassemble(acp_pdu_type *rx_pdu,processvars_type *processdata)
{
 unsigned int bytes_length;
 unsigned char *new_data;
 acp_pdu_type *tx_pdu;

 tx_pdu = (acp_pdu_type *) malloc(sizeof(acp_pdu_type));
 bytes_length = get_bytes_length(processdata->insequenceq);
 new_data = (unsigned char *) malloc(sizeof(unsigned char) * bytes_length);
 build_sequence(processdata->insequenceq,new_data,bytes_length);                    10
 tx_pdu = build_data_indication_pdu(rx_pdu->sapi,new_data,btol3(bytes_length));

 put(processdata->nlid,ACP_DATA_indication,tx_pdu);

 return;
}

void put(procid_type procid,signals signal_name,acp_pdu_type *the_data)
{
 Ros_message *send_buf;                                                             20
 POOL_ID send_pool_id;
 send_pool_id = Get_Pool_Id(sizeof(Ros_message) - sizeof(BUF_HEADER));
 send_buf = (Ros_message *) Get_Buf(send_pool_id);
 send_buf->signal = signal_name;
 send_buf->value = (void *) the_data;
 send_buf->instance = 0;
 if (procid == sap10)
   {
    Put_Msg((BUF_PTR) send_buf, Get_Task_Id(SAP10));
   }                                                                                30
 else if (procid == sap20)
   {
    Put_Msg((BUF_PTR) send_buf, Get_Task_Id(SAP20));
```

```
    }
  else if (procid == sap13)
    {
      Put_Msg((BUF_PTR) send_buf, Get_Task_Id(SAP13));
    }
  else if (procid == sap23)
    {
      Put_Msg((BUF_PTR) send_buf, Get_Task_Id(SAP23));
    }
  else if (procid == nl1)
    {
      Put_Msg((BUF_PTR) send_buf, Get_Task_Id(L31));
    }
  else if (procid == nl2)
    {
      Put_Msg((BUF_PTR) send_buf, Get_Task_Id(L32));
    }
  else if (procid == multiplex1)
    {
      Put_Msg((BUF_PTR) send_buf, Get_Task_Id(MULTIPLEX1));
    }
  else if (procid == multiplex2)
    {
      Put_Msg((BUF_PTR) send_buf, Get_Task_Id(MULTIPLEX2));
    }
  else if (procid == mac)
    {
      Put_Msg((BUF_PTR) send_buf, Get_Task_Id(MAC));
    }
  return;
}


unsigned int get_bytes_length(queue_type1 *insequenceq)
{
  unsigned int byte_length=0;
  data_type1 *head_of_list;

  head_of_list = insequenceq->head;

  while (head_of_list!=0)
    {
      byte_length = byte_length + acptob(head_of_list->queue_data->length);
      head_of_list = head_of_list->next_data;
    }

  return byte_length;
}
```

7.6.3 MOUSETRAP Generated Code

```
void Reassemble(struct _bsqrep var9, struct ProcessVars_Type * var10){
        struct _bsqrep var2 ;
        unsigned int var3 ;
        struct ByteSequence_Type * var4 ;
        struct ByteSequence_Type * var5 ;
        unsigned int var6 ;
        unsigned int var7 ;
        unsigned int var8 ;
        struct _bsqrep t46 ;
        struct _bsqrep tmp42 ;                                                  10
        struct _bsqrep t47 ;
        struct _bsqrep tmp43 ;
        struct ByteSequence_Type * t48 ;
        int t49 ;
        struct ByteSequence_Type * tmp44 ;
        struct ByteSequence_Type * t50 ;
        struct ByteSequence_Type * t51 ;
        struct ByteSequence_Type * tmp45 ;
        struct _nodePtr * list11 ;
        struct IPDUQueue_Type * var1 ;                                          20
        var4 = (struct ByteSequence_Type *) malloc(sizeof(struct ByteSequence_Type)) ;
        var4 -> tag = 0 ;
        list11 = var10 -> InSequenceQ -> qlist ;
        var3 = 0 ;
        while(list11 -> cdr != 0) {
                var1 = list11 -> cdr -> car ;
                var7 = acptob(var1 -> length) ;
                var3 += var7 ;
                list11 = list11 -> cdr ;
        }                                                                       30
        var6 = btol3(var3) ;
        var8 = l3tob(var6) ;
        tmp45 = var4 ;
        t50 = tmp45 ;
        t51 = init_sequence(t50, var6, var8) ;
        var5 = build_sequence(t51, var6, var10) ;
        tmp44 = var5 ;
        t48 = tmp44 ;
        t49 = var9. _bitmem[0] & 0x7 ;
        var2 = build_data_indication_pdu(t49, t48, var6) ;                      40

        switch(var10 -> NetworkLayerId) {
        case 2:
                tmp43 = var2 ;
                t47 = tmp43 ;
                {
                        Ros_message * send_buf ;
                        POOL_ID send_pool_id ;
                        send_pool_id = Get_Pool_Id(sizeof(Ros_message) - sizeof(BUF_HEADER)) ;
```

```
                send_buf = (Ros_message *) Get_Buf(send_pool_id) ;                    50
                send_buf -> signal = 15 ;
                send_buf -> value.bitseq = t47 ;
                send_buf -> instance = 0 ;
                Put_Msg((BUF_PTR) send_buf, Get_Task_Id(L31)) ;
        }
        break ;
    case 1:
        tmp42 = var2 ;
        t46 = tmp42 ;
        {                                                                            60
                Ros_message * send_buf ;
                POOL_ID send_pool_id ;
                send_pool_id = Get_Pool_Id(sizeof(Ros_message) - sizeof(BUF_HEADER)) ;
                send_buf = (Ros_message *) Get_Buf(send_pool_id) ;
                send_buf -> signal = 15 ;
                send_buf -> value.bitseq = t46 ;
                send_buf -> instance = 0 ;
                Put_Msg((BUF_PTR) send_buf, Get_Task_Id(L32)) ;
        }
        break ;                                                                      70
    }
    _free(var9. _bitmem) ;
    return  ;
}
```

7.6.4 Readability

In terms of readability, the specifications are easiest to understand. However, the code that was written by hand is much more readable than the code that is generated by MOUSETRAP. This is true mainly because the variable names in the handwritten code have been chosen carefully to have meaning and to follow the specifications. Also, the generated code is longer than the handwritten code.

The code that is generated by the transformer is much harder to read because all of the variables have generated names which have no significance to their function in the code. In addition, from this example, it can be seen that the code to send a message in ROS has been inlined in the example from the generated code where there is a function in the manually written code. It is these functions that also make the manually written code much easier to follow.

So, it is apparent, that the generated code is extremely difficult to read. However, the code that is generated is not meant to be read by programmers. It is intended that developers should solely look at the specifications.

The specifications are initially as easy to read as the handwritten C code, provided that the engineer coding the specifications closely follows the original specifications. Although not illustrated in this example, the specifications can sometimes express some complicated C expressions through useful abstractions which make the specifications clearer. One example of this is the *statemachine* construct which is available to the user in the specification language but not in C. Additionally, if the algorithm were much more complicated, then the C code might lose some clarity which the specifications would preserve. Also, as the handwritten code gets modified, it will become harder to read the pure C code.

7.6.5 Extensibility

Finally, in terms of extensibility, expanding the specifications is far easier to expand than the handwritten code. Regardless, of whether code is generated manually or through program transformation, understanding another person's code tends to be

tedious and time consuming. Therefore, expanding the code generated by MOUSETRAP is substantially easier, because the actual code never has to be modified. A developer would simply modify the specifications, which are substantially easier to understand.

Chapter 8

Conclusions

8.1 Advantages of Using MOUSETRAP as a Development Tool

The initial benefits of program transformation can be seen by just looking at the potential savings in the development time. However, there are many other advantages to using program transformation for software development.

The initial savings in using program transformation in this case was a month. The two month development time using program transformation included the time to encapsulate the knowledge of ROS into the transformer system. So, if anyone wanted to develop code in this environment at a later date, the development time would essentially be the time to design the specifications. This time would be needed if the programmer was using traditional methods or program transformation. Additionally, at the end of the development, the specifications, the documentation of the code, are already finished.

Program transformation also allows standardization of code. If there are multiple engineers working on a project, it is extremely difficult to standardize the code that is written because each engineer may have his own coding style and conventions. Additionally, it is even more difficult to modify the code if there are any changes in the design of the code. Program transformation allows the code to look uniform, and

if there is a change in the design, one only needs to change the specifications and run the specifications through the transformer system.

Another benefit of program transformation during the development process is the standardization of errors. A mistake in the generated code indicates a mistake in the specifications or in one of the transformer rules. Mistakes in the specifications are much easier to identify than mistakes in C code. Searching the specifications for algorithmic errors is much easier than identifying errors in the handwritten code. If the handwritten code is not extremely clear, then determining the developer's intention can be difficult and finding the error will require additional time. Locating errors in the transformation rules, however, is much more difficult than finding errors in the specification. This requires looking at the generated code and trying to isolate the point at which an error was introduced. Once the rule is found, however, that rule just needs to be corrected, and the code can be regenerated. This guarantees that all errors of that class will be fixed.

Finally, MOUSETRAP allows for easier maintenance of code. This is an important advantage for companies that are constantly producing new versions of their software. Engineers will no longer have to decipher another software engineer's code but will instead be able to modify the specifications. This results in more reliable software as well as the assurance that documentation will always keep pace with the software.

8.2 Disadvantages of Using MOUSETRAP as a Development Tool

There are, however, disadvantages to using the MOUSETRAP system. First, it is extremely important to have a strong understanding of the specification language. It takes some significant time and effort to learn how to write correct specifications. Also, the specifications must be extremely detailed when using MOUSETRAP, which is not always necessary if the coding is done manually.

Additionally, unless the specifications can initially be tested for correctness, one

has to wait until all of the code is generated to see if the system is actually exhibiting correct behavior.[1] This can be tedious and is no different from debugging handwritten code. The issue is that the user has to debug the generated code, which is difficult to understand, and if errors are found, the generate and test cycle takes too long because the transformer system is slow. This is because the specifications can not be executed, and runtime debugging tools can not be used.

Another disadvantage was the time it took to learn how to add rules to the system. Even if someone knew the details of a particular platform, it would take some training to learn how to write correct MOUSETRAP rules. Additionally, the system interface was difficult to navigate because of its assumption that the users were knowledgeable about MOUSETRAP.

Other disadvantages specifically related to the MOUSETRAP system include the lack of stability of the system and resulting slowness. The specifications had to be broken up into several pieces and run on several machines simultaneously for reasonable results. Data was not recorded on the time necessary to generate each of the individual pieces. If this data was obtained, it would be useful to determine a relation between specification code size and the time to generate the code.

Finally, because the system is new, there were several mistakes in the rules mainly because many unexpected cases were encountered while running the specifications on MOUSETRAP. This slowed down progress a great deal.

8.3 Conclusions

Program transformation is an effective way to encapsulate knowledge so that it is easier for people to develop reliable code. The advantages of faster development time and ease of maintenance is of great value in commercial software development. Additionally, writing specifications independent of implementation will reduce errors and therefore produce more reliable software.

It is essential, however, that there is some method to validate specifications for

[1]Since completing my research, a specification validation environment has been developed.

MOUSETRAP to be an effective tool. Additionally, learning how to write specifications and the rules for the transformer system takes a significant amount of time.

Overall, MOUSETRAP can be used to develop efficient commercial software with performance that is similar to handwritten code. However, the speed, stability, and the interface of MOUSETRAP needs to improve before it is a feasible method of large scale commercial software development.

Appendix A

Testing and Simulation

The initial goal was to be able to test the code on the actual hardware of the radio. However, due to the lack of access to an appropriate test environment of the actual hardware, a simulation of the hardware environment was used to test the code.

A.1 Hardware

The hardware platform that the code will be used with is a radio that uses a HC16 microprocessor and runs ROS, Radio Operating System. This is a standard platform that most radios at Motorola use.

A.2 XROS

XROS is a program that many of the product groups at Motorola use to develop ROS software for the radios. The XROS program runs under XWindows and simulates the ROS environment. This was unfortunately not a viable option for simulating my code because XROS simulates each ROS task as a process and therefore pointers between ROS Tasks could not be sent. The code that was generated simulated a ROS message as a structure that contained a pointer to pass data.

A.3 PPCR

Finally, I used a package call PPCR which was developed at Xerox PARC. PPCR stands for Posix Portable Common Runtime (PPCR). It provides provides "integrated user-level support for pre-emptive lightweight threads, garbage collected storage, and dynamic programming loading." Since PPCR is a threads package, I wrote a set of interface functions that would allow ROS to be simulated under this environment. Each ROS task is just modeled as a thread so that pointers can be sent between tasks.

A.4 Interface Functions

```
#define Delay(ticks) \
OS_WAKE_EVERY(ticks); \
OS_WAKE_POINT(ticks);

#define Put_Msg(message,dest_tid) \
ROS_Put_Msg(message,dest_tid,NO_REPLY_REQUEST,STANDARD);

#define Get_Msg(t_name) \
ROS_Get_Msg(t_name);
                                                                    10

POOL_ID Get_Pool_Id(unsigned int size)
{
 extern pool_node_t pool_array[];
 unsigned int i;
 POOL_ID retval=0;
 for (i=0;i<NUM_POOLS;i++)
  {
   if ((pool_array[i].buffer_size)==size)
        {                                                           20
         retval=pool_array[i].pool_id;
         break;
        }
  }
 return retval;
}

TASK_ID Get_Task_Id(unsigned int name)
{
 extern task_node_t task_array[];                                   30
 unsigned int i;
 TASK_ID retval=0;
```

```
 for (i=0;i<NUM_TASKS;i++)
  {
   if ((task_array[i].task_name)==name)
        {
         retval=(task_array[i].task_id);
         break;
        }
  }
 return retval;
}

BUF_PTR Get_Buf(POOL_ID poolid)
{
 extern pool_node_t pool_array[];
 unsigned int i;
 BUF_PTR retptr=0;
 for (i=0;i<NUM_POOLS;i++)
  {
   if ((pool_array[i].pool_id)==poolid)
        {
         if (pool_array[i].count==0)
           printf("Get_Buf error:buffer pool exhausted!\n");
         else
           {
            retptr=(BUF_PTR) malloc(sizeof(BUF_HEADER) +
                                    (pool_array[i].buffer_size));
            pool_array[i].count=pool_array[i].count - 1;
/*           printf("Get_Buf:count of pool_array is %d!\n",pool_array[i].count);*/
           }
         break;
        }
  }

 retptr->pool_id = poolid;
 return retptr;
}

void Free_Buf(BUF_PTR bufptr)
{
 extern pool_node_t pool_array[];
 POOL_ID poolid;
 unsigned int i;

 poolid = bufptr -> pool_id;
 free(bufptr);
 for (i=0;i<NUM_POOLS;i++)
  {
   if ((pool_array[i].pool_id)==poolid)
        {
/*         printf("Free_Buf: freeing a buffer with poolid = %d!\n",poolid);*/
         pool_array[i].count=pool_array[i].count + 1;
/*         printf("Free_Buf: count of pool_array is %d!\n",pool_array[i].count);*/
         break;
```

```
        }
    }
  return;                                                          90
}


void ROS_Put_Msg(BUF_PTR message,TASK_ID dest_taskid,
                 reply_bit_t reply_bit,delivery_bit_t delivery_bit)
{
  extern task_queue_node_t task_queue_array[];
  Qid *tmpqid=0;
  Qid send_queue;
  int message_size=0,i=0;                                          100

  message->reply_id=reply_bit;
  message->delivery_id=delivery_bit;
  message_size=sizeof(*(message));
  for (i=0;i<NUM_POOLS;i++)
    {
     if ((pool_array[i].buffer_size)==message_size)
          {
           message->pool_id=pool_array[i].pool_id;
           break;                                                  110
          }
    }

  tmpqid = task_queue_array[(dest_taskid -1)].queue_id;
  send_queue = *(tmpqid);

  if ((reply_bit==NO_REPLY_REQUEST) && (delivery_bit==STANDARD))
    {     OS_SEND_MESSAGE(send_queue,GREETING,message);
    }
  return;                                                          120
}


BUF_PTR ROS_Get_Msg(unsigned int task_name)
{
  TASK_ID receive_task_id;
  extern task_queue_node_t task_queue_array[];
  Qid *tmpqid=0;
  Qid receive_queue;
  void *message;                                                   130
  int opcode;

  receive_task_id = Get_Task_Id(task_name);
  tmpqid = task_queue_array[(receive_task_id -1)].queue_id;
  receive_queue = *(tmpqid);
  OS_RECEIVE_MESSAGE(receive_queue,opcode,message);
  return ((BUF_PTR) message);
}
```

Appendix B

Scheduler Tool Source Code

```
(defvar *goto-c* nil
  "Global storage for information about rules.")

(defstruct rule-set
  name
  pre-conditions
  post-conditions
  language-layer
  cost)

(defmacro defprop (&key name pre-conditions post-conditions
language-layer cost)
  `(setf *goto-c* (cons
   (make-rule-set :name ,name
  :pre-conditions ,pre-conditions
  :post-conditions ,post-conditions
  :language-layer ,language-layer
  :cost ,cost)
   *goto-c*)))

(defun rule-list-for-layer-n (layernumber ruledatabaselist)
  (mapcan #'(lambda(ruledata)
         (if (inlist-p layernumber
    (rule-set-language-layer ruledata))
  (list ruledata)))
  ruledatabaselist))
```

```
(defun create-layer-list (layernumber ruledatabaselist)
  (if (equal layernumber 5)
      nil
    (let ((rulelistlayern (rule-list-for-layer-n
   layernumber ruledatabaselist)))
      (if (not (null rulelistlayern))
  (cons (rule-list-for-layer-n layernumber ruledatabaselist)
(create-layer-list  (+ layernumber 1) ruledatabaselist))
(create-layer-list (+ layernumber 1) ruledatabaselist)))))

(defun extract-all-rule-names (listofrulesdata)
  (mapcar #'(lambda(ruledata)
      (rule-set-name ruledata))
  listofrulesdata))

(defun inlist-p (element listtocheck)
  (subsetp (list element) listtocheck))

(defun first-path-used-all-rules-p (listofpaths listofrulenames)
  (let ((first-path-rule-names (first (first listofpaths))))
    (subsetp listofrulenames first-path-rule-names)))


(defun first-path-reached-conditions-p (listofpaths conditionsneededtostop)
  (let ((first-path-conditions (second (first listofpaths))))
    (subsetp conditionsneededtostop first-path-conditions)))

(defun get-rules-not-used-data (listofrulesdata rulesusedlist)
  (mapcan #'(lambda(ruledata)
      (if (not (inlist-p (rule-set-name ruledata) rulesusedlist))
  (list ruledata)))
  listofrulesdata))


(defun screws-up-1 (postconds
    precondsnotused
    postcondsallrules)
  (let ((list1 (intersection postconds precondsnotused))
(list2 (intersection postconds postcondsallrules)))
    (and (not (null list1))
(not
 (and (equal (length list1) (length list2))
      (subsetp list1 list2))))))

(defun get-last-rule-data (rulesusedlist rulesdata)
```

```
  (let ((lastrulename (first (last rulesusedlist))))
    (first
     (mapcan #'(lambda(ruledata)
 (if (equal (rule-set-name ruledata) lastrulename)
     (list ruledata)))
     rulesdata))))

(defun direct-link-p (ruledata lastruledata)
 (if (equal lastruledata nil)
     t
   (not (null (append
       (intersection (extract-true-conditions
      (rule-set-pre-conditions ruledata))
     (extract-true-conditions
      (rule-set-post-conditions lastruledata)))
       (intersection (extract-false-conditions
      (rule-set-pre-conditions ruledata))
     (extract-false-conditions
      (rule-set-post-conditions lastruledata)))))))))

(defun filter-neighbors-2-aux (lastruleuseddata
       possneighbors
       rulesusedlist
       rulesdata
       origlist
       notusedlist
       notnulllist
       directlinklist)

  (if (null possneighbors)
     (if (not (null directlinklist))
 directlinklist
       (if (not (null notnulllist))
   notnulllist
 (if (not (null notusedlist))
     notusedlist
   origlist)))
    (let ((notused (not (inlist-p (rule-set-name (first possneighbors))
  rulesusedlist)))
  (notnull (not (null (rule-set-pre-conditions (first possneighbors)))))
  (directlink (direct-link-p (first possneighbors) lastruleuseddata)))
      (filter-neighbors-2-aux
       lastruleuseddata
       (rest possneighbors)
       rulesusedlist
```

```
      rulesdata
      origlist
      (if notused
  (cons (first possneighbors) notusedlist)
notusedlist)
      (if (and notused notnull)
  (cons (first possneighbors) notnulllist)
notnulllist)
      (if (and notused notnull directlink)
  (cons (first possneighbors) directlinklist)
directlinklist)))))



(defun filter-neighbors-2 (possneighbors rulesusedlist rulesdata)
  (let ((lastruleuseddata (get-last-rule-data rulesusedlist rulesdata)))
    (filter-neighbors-2-aux lastruleuseddata
    possneighbors
    rulesusedlist
    rulesdata
    possneighbors
    '()
    '()
    '())))



(defun get-neighbors-2-aux (rulesdata currentstatelist rulesusedlist)
  (let ((all-rules-post-conditions (find-all-rules-post-conds rulesdata)))
    (let ((postcondsfalseallrules (extract-false-conditions
   all-rules-post-conditions))
  (postcondstrueallrules (extract-true-conditions
  all-rules-post-conditions)))
      (let ((possibleneighbors
     (mapcan #'(lambda(ruledata)
 (let ((precondstrue (extract-true-conditions
      (rule-set-pre-conditions ruledata)))
       (precondsfalse (extract-false-conditions
       (rule-set-pre-conditions ruledata)))
       (postcondstrue (extract-true-conditions
       (rule-set-post-conditions ruledata)))
       (postcondsfalse (extract-false-conditions
       (rule-set-post-conditions ruledata))))
   (if (and
(and (subsetp precondstrue currentstatelist)
```

```
     (null (intersection precondsfalse
 currentstatelist)))
(or (not (subsetp postcondstrue currentstatelist))
    (not (null (intersection postcondsfalse
     currentstatelist)))))
       (let ((precondsrulesnotused
      (find-all-rules-pre-conds
       (get-rules-not-used-data
rulesdata
(cons (rule-set-name ruledata)
      rulesusedlist)))))
 (let ((precondsfalsenotused
(extract-false-conditions
 precondsrulesnotused))
       (precondstruenotused
(extract-true-conditions
 precondsrulesnotused)))
   (if (and
(not
 (screws-up-1
  postcondstrue
  precondsfalsenotused
  postcondsfalseallrules))
(not
 (screws-up-1
  postcondsfalse
  precondstruenotused
  postcondstrueallrules)))

       (list ruledata)))))))
 rulesdata)))
     (filter-neighbors-2 possibleneighbors rulesusedlist rulesdata)))))

(defun get-neighbors-2 (rulesdata currentstatelist rulesusedlist)
  (let ((neighborlist (get-neighbors-2-aux rulesdata currentstatelist
   rulesusedlist)))
    (if (null neighborlist)
(list (list 'END-PATH '() '0))
      (mapcar #'(lambda(neighbordata)
  (list
   (rule-set-name neighbordata)
   (rule-set-post-conditions neighbordata)
   (rule-set-cost neighbordata)))
      neighborlist))))
```

```
(defun change-state (currentstatelist newpostconditions)
  (let ((true-list
 (mapcan #'(lambda (x)
     (if (equal (second x) 'true)
 (list (first x))))
 newpostconditions))
(false-list
 (mapcan #'(lambda (x)
     (if (equal (second x) 'false)
 (list (first x))))
 newpostconditions))
(combined-list)
(toadd-list))
    (setf combined-list
  (mapcan #'(lambda(postcondname)
      (if (or (inlist-p postcondname true-list)
      (not (inlist-p postcondname false-list)))
  (list postcondname)))
  currentstatelist))
    (setf toadd-list
  (mapcan #'(lambda(postcondname)
      (if (not (inlist-p postcondname combined-list))
  (list postcondname)))
  true-list))
    (append combined-list toadd-list)))

(defun create-new-paths (rulesusedlist currentstatelist
       costsofarlist
       costremaininglist
       newneighborslist)
   (mapcar #'(lambda (neighbordata)
      (list (append rulesusedlist (list (first neighbordata)))
    (change-state currentstatelist
  (second neighbordata))
    (list (+ (third neighbordata) (first costsofarlist)))
    (if (not (inlist-p (first neighbordata) rulesusedlist))
(list (- (first costremaininglist) (third neighbordata)))
      (list (first costremaininglist)))))
    newneighborslist))



(defun get-new-paths (rulesdata listofpaths listofrulenames)
```

```
  (let ((first-path-rule-names (first (first listofpaths)))
(first-path-current-state (second (first listofpaths)))
(first-path-cost-so-far (third (first listofpaths)))
(first-path-cost-remaining (fourth (first listofpaths))))
    (let ((creatednewpaths (create-new-paths
    first-path-rule-names
    first-path-current-state
    first-path-cost-so-far
    first-path-cost-remaining
    (get-neighbors-2
     rulesdata
     first-path-current-state
     first-path-rule-names))))
      creatednewpaths)))

(defun get-left-list (restoflist ordering-function pivot)
  (mapcan #'(lambda(listdata)
      (if (funcall ordering-function listdata pivot)
  (list listdata)))
  restoflist))

(defun get-right-list (restoflist ordering-function pivot)
  (mapcan #'(lambda(listdata)
      (if (funcall ordering-function listdata pivot)
  (list listdata)))
  restoflist))

(defun quick-sort-paths (list-to-sort)
  (if (or (null list-to-sort) (null (rest list-to-sort)))
      list-to-sort
    (let ((pivot (first list-to-sort))
  (restoflist (rest list-to-sort)))
      (let ((leftlist (get-left-list  restoflist #'ordering-function-left pivc
       (rightlist (get-right-list restoflist #'ordering-function-right pivot))
(let ((sortedleft (quick-sort-paths leftlist))
      (sortedright (cons pivot (quick-sort-paths rightlist))))
  (append sortedleft sortedright))))))

(defun sum-list (list-to-sum)
  (if  (null list-to-sum)
      0
    (+
     (if (null (first list-to-sum))
 0
       (first list-to-sum))
```

```
     (sum-list (rest list-to-sum)))))

(defun ordering-function-left  (path1 path2)
  (let ((path1-cost-so-far (first (third path1)))
(path2-cost-so-far (first (third path2)))
(path1-underestimate-of-rest (first (fourth path1)))
(path2-underestimate-of-rest (first (fourth path2))))
    (or (< (+ path1-cost-so-far path1-underestimate-of-rest)
   (+ path2-cost-so-far path2-underestimate-of-rest))
(and (= (+ path1-cost-so-far path1-underestimate-of-rest)
(+ path2-cost-so-far path2-underestimate-of-rest))
     (>= path1-cost-so-far path2-cost-so-far)))))


(defun ordering-function-right  (path1 path2)
  (let ((path1-cost-so-far (first (third path1)))
(path2-cost-so-far (first (third path2)))
(path1-underestimate-of-rest (first (fourth path1)))
(path2-underestimate-of-rest (first (fourth path2))))
    (or (> (+ path1-cost-so-far path1-underestimate-of-rest)
   (+ path2-cost-so-far path2-underestimate-of-rest))
(and (= (+ path1-cost-so-far path1-underestimate-of-rest)
(+ path2-cost-so-far path2-underestimate-of-rest))
     (< path1-cost-so-far path2-cost-so-far)))))


(defun end-path-p (list-of-stuff)
  (let ((reversedlist (reverse list-of-stuff)))
    (equal (first reversedlist) 'END-PATH)))

(defun conditions-same-p (conds1 conds2)
  (and (equal (length conds1) (length conds2))
       (subsetp conds1 conds2)))

(defun equal-path-p (path1 path2)
  (and  (subsetp (first path2)
 (first path1))
(conditions-same-p (second path1) (second path2))
(<= (first (third path1)) (first (third path2)))))

(defun lessthan-path-p (path1 path2)
  (and  (subsetp (first path2)
 (first path1))
(subsetp (second path2) (second path2))))
```

```
(defun take-out-redundant-paths-2 (newpaths currentpaths)
  (if (not (null newpaths))
      (take-out-redundant-paths-2
       (rest newpaths)
       (mapcan #'(lambda (currentpathdata)
   (if
       (and (not (equal-path-p (first newpaths) currentpathdata))
    (not (lessthan-path-p (first newpaths) currentpathdata)))
       (list currentpathdata)))
       currentpaths))
    currentpaths))

(defun take-out-redundant-paths-aux (listofpaths)
  (if (null listofpaths)
      nil
    (let ((useless-path (useless-to-rest-p (first listofpaths) (rest listofpat
      (if (not useless-path)
  (cons (first listofpaths)
(take-out-redundant-paths-aux (rest listofpaths)))
(take-out-redundant-paths-aux (rest listofpaths))))))


(defun eliminate-dead-ends (listofpaths)
  (mapcan #'(lambda(pathdata)
      (let ((path-rule-names (first pathdata)))
(if (not (end-path-p path-rule-names))
    (list pathdata))))
  listofpaths))

(defun find-paths-aux (rulesdata
       listofrulenames
       nextlayerconds
       listofpaths
       )
  (if (and (first-path-reached-conditions-p  listofpaths nextlayerconds)
   (equal (first (fourth (first listofpaths))) 0))
      (first listofpaths)
    (let ((newpaths (get-new-paths rulesdata listofpaths listofrulenames)))
      (find-paths-aux rulesdata
      listofrulenames
      nextlayerconds
      (quick-sort-paths
       (append
newpaths
```

```
(take-out-redundant-paths-2
 newpaths
 (rest listofpaths))))))))

(defun sum-of-rules-cost (rulesdata)
  (sum-list
   (mapcar #'(lambda(ruledata)
       (rule-set-cost ruledata))
   rulesdata)))

;;three elements in every path entry
;;rule-set-name list
;;cost so far
;;cost to go (sum of all only one application rules)
;;conditions which are true after this sequence of rule-sets

(defun find-paths (rulesdata listofcurrentconds nextlayerconds)
  (let ((listofpossiblepaths (list
      (list '() listofcurrentconds
    '(0)
    (list (sum-of-rules-cost rulesdata))
    )))
(listofrulenames (extract-all-rule-names rulesdata)))
    (find-paths-aux rulesdata
    listofrulenames
    nextlayerconds
    listofpossiblepaths)))

(defun eliminate-duplicates (list1)
  (if (null list1)
      nil
    (if (inlist-p (first list1) (rest list1))
(eliminate-duplicates (rest list1))
      (cons (first list1)
    (eliminate-duplicates (rest list1))))))

(defun fold-list (list1)
  (if (null list1)
      nil
    (append (first list1)
    (fold-list (rest list1)))))

(defun find-all-rules-pre-conds (ruleslist)
  (fold-list (mapcar #'(lambda(ruledata)
      (rule-set-pre-conditions ruledata))
```

```
      ruleslist)))

(defun find-all-rules-post-conds (ruleslist)
  (fold-list (mapcar #'(lambda(ruledata)
      (rule-set-post-conditions ruledata))
  ruleslist)))

(defun extract-true-conditions (conditionslist)
  (eliminate-duplicates
   (mapcan #'(lambda(conddata)
       (if (equal (second conddata) 'true)
   (list (first conddata))))
   conditionslist)))

(defun extract-false-conditions (conditionslist)
  (eliminate-duplicates
   (mapcan #'(lambda(conddata)
       (if (equal (second conddata) 'false)
   (list (first conddata))))
   conditionslist)))

(defun find-next-layer-conds (nextlayerlist)
  (if (null nextlayerlist)
      nil
    (let ((all-rules-pre-conds (find-all-rules-pre-conds nextlayerlist))
  (all-rules-post-conds (find-all-rules-post-conds nextlayerlist)))
      (let ((pre-conditions-true-list (extract-true-conditions
       all-rules-pre-conds))
    (post-conditions-true-list (extract-true-conditions
all-rules-post-conds)))
(mapcan #'(lambda (conddata)
    (if (not (inlist-p conddata post-conditions-true-list))
(list conddata)))
pre-conditions-true-list)))))


(defun find-the-schedule-aux (layerlist currentcondslist)
  (if (null layerlist)
      nil
    (let ((next-layers-conds (find-next-layer-conds (second layerlist))))
      (let ((this-layers-path (find-paths (first layerlist)
  currentcondslist
  next-layers-conds)))
(let ((next-set-current-conds (second this-layers-path)))
  (cons this-layers-path
```

```
(find-the-schedule-aux (rest layerlist)
       next-set-current-conds)))))))

(defun find-the-schedule (ruledatabaselist currentconds)
  (let ((layerlist (create-layer-list 0 ruledatabaselist))
(listofcurrentconds currentconds))
    (find-the-schedule-aux layerlist listofcurrentconds)))

(defun testfind ()
  (find-the-schedule *goto-c* '(symbols-table datatype-table
      datatype-new-table type-rels-table global-vars-table)))


(defun testfinddata ()
     (find-the-schedule *goto-c* '()))
```

Appendix C

ROS System Knowledge Base

C.1 Rule Sets

C.1.1 Rule Set 1

```
pre-back-end1 is grammar cmod;

?signal: Expr;
?value: Expr;
?instance: Expr;
?sample,?el1:Exprlist;
?se:Expr;
?sid:Ident;

?queuename:Ident;                                                    10
?taskid:= <Ident>'x';
?queueid :=<Ident>'x';
?qid2 :=<Ident>'x';
?tempid :=<Ident>'x';
?dummy1 :=<Ident>'x';
?procname :=<Ident>'x';


/. <Expr>'OS_SEND_MESSAGE (?queuename,?signal,?value, ?instance)'
 & ?tempid := ?queuename                                             20
 & ?dummy1 := new <Ident> 'queue'
  & ?sample := <Exprlist>' '
  & ?se := <Expr>'?sid'
 & ?el1 := eval* table-keys-as-list ?sample ?se ?sid |*process-table*|
 & ?el1 := ?el1 /. <Expr>'?taskid'
                          & ?queueid := eval* get-from-table-if-none
                            ?taskid |*process-table*| ?dummy1
```

```
                    & ?qid2 := ?queueid as cmod::Ident
                    & ?qid2 == ?tempid
                    & ?procname := eval string-upcase ?taskid                    30
                ==><Expr>'?taskid'
==> <Expr>'OS_SEND_MESSAGE (?procname,?signal,?value, ?instance)'
```

C.1.2 Rule Set 2

```
pre-back-end2 is grammar cmod;

?procname, ?dummy, ?taskid: Ident;
?signal: Expr;
?value: Expr;
?instance: Expr;

/. <Expr>'OS_SEND_MESSAGE (?procname,?signal,?value, ?instance)'

  & ?dummy := eval get-first-available-taskid |*task-ids*|                       10
  & ?taskid := eval get-from-table-if-none-s ?procname |*process-id-table*| ?dummy
  & ! ?taskid == ?dummy
==> <Expr>'OS_SEND_MESSAGE (?procname, ?signal, ?value, ?instance)'
; # taskid name is already in table

/. <Expr>'OS_SEND_MESSAGE (?procname,?signal,?value, ?instance)'
  & ?dummy := eval get-first-available-taskid |*task-ids*|
  & ?taskid := eval get-from-table-if-none-s  ?procname |*process-id-table*| ?dummy
  & ?taskid == ?dummy
  & test put-into-table-s ?procname ?taskid |*process-id-table*|               20
  & test update-taskid-list |*task-ids*|
==> <Expr>'OS_SEND_MESSAGE (?procname, ?signal, ?value, ?instance)'
; # taskid name is generated and inserted into table
```

C.1.3 Rule Set 3

```
delay-every is grammar cmod;

?time:Expr;

/. <Expr>'_delay_every(?time)'
==> <Expr>'Delay(?time)'
```

C.1.4 Rule Set 4

```
send-C-expand is grammar cmod;
```

```
?taskid : Ident;
?newtaskid : Ident;
?signal : Expr;
?value : Expr;
?instance : Expr;
?newchar : Charconst;
?procname:Ident;
                                                                          10
/.
  <Stmt>'OS_SEND_MESSAGE(?procname,?signal,?value, ?instance);'
        & ?newtaskid := eval string-upcase ?procname
    ==>
  <Stmt>'{
  Ros_message * send_buf;
  POOL_ID send_pool_id;
  send_pool_id = Get_Pool_Id((sizeof(Ros_message) - sizeof(BUF_HEADER)));
  send_buf = (Ros_message *) Get_Buf(send_pool_id);
                                                                          20
  send_buf->signal = ?signal;
  send_buf->value = ?value;
  send_buf->instance = ?instance;

  Put_Msg((BUF_PTR) send_buf,Get_Task_Id(?newtaskid));
    }' (typedef POOL_ID, Ros_message, BUF_HEADER,BUF_PTR)
```

C.1.5 Rule Set 5

```
ros-delete-thingy is grammar cmod;

?var, ?data: Ident;
?decll: Decllist;
?stmtl: Stmtlist;

/.  <Unit>'void delete_thingy (void ?data) { ?decll ?stmtl }'
  & ?stmtl := ?stmtl
                    /. <Expr>'_get_input_queue ()'
                         ==>                                              10
                       <Expr>'_get_input_queue (?data)'
                         ;
                         # expand _get_signal_name()
                /. <Expr>'_get_signal_name()'
                     ==>
                  <Expr>'_get_signal_name (?data)'
                  ;
                         # expand _get_instance_number();
                 /. <Expr>'_get_instance_number()'
                     ==>                                                  20
                   <Expr>'_get_instance_number (?data)'
                  ;

==> <Unit>'void delete_thingy (Ros_message * ?data) { ?decll ?stmtl }' (typedef Ros_message)
```

;

C.1.6 Rule Set 6

```
ros-get-input-queue is grammar cmod;
?ul : Unitlist;
?typeid : Type;
?fname : Ident;
?parmlist : Parameterlist;
?decllist : Decllist;
?sl : Stmtlist;
?m : Ident;
?var : Ident;
                                                                        10
/. <Func>'
?typeid  ?fname ( ?parmlist )
{ ?decllist
  ?sl
}'
  & <Stmt>'_get_input_queue();' <= ?sl
  & ?m  := new <Ident>'m' #since we have a _get intro a msg var
  & ?sl := ?sl /. <Stmt>'_get_input_queue();'
            ==>
              <Stmt>'{ ?m = (Ros_message *) Get_Msg();}' (typedef Ros_message);   20
                        # expand _get_signal_name()
               /. <Expr>'_get_signal_name()'
                    ==>
                 <Expr>'_get_signal_name (?m)'
                 ;
                        # expand _get_instance_number();
               /. <Expr>'_get_instance_number()'
                    ==>
                 <Expr>'_get_instance_number (?m)'
                 ;                                                          30
                        # expand get_value
               /. <Stmt>'?var = _get_value(?var);'
                    ==>
                        <Stmt>'{?var = _get_value(?m,?var);
                                Free_Buf((BUF_PTR) ?m);}' (typedef BUF_PTR)
                                                          ;
                  /. <Stmt>'delete_thingy ();'
                       ==>
                    <Stmt>'{
                                delete_thingy (?m);                         40
                                Free_Buf((BUF_PTR) ?m);}'(typedef BUF_PTR)
                     ;
==> <Func>
'?typeid ?fname ( ?parmlist )
 { ?decllist
   Ros_message * ?m;
   ?sl
```

}‘ (typedef Ros_message)
;

C.1.7 Rule Set 7

```
ros−get−slots is grammar cmod;

?var, ?m: Ident;
?e : Expr;

/. <Expr>‘_get_signal_name (?m)‘
==> <Expr>‘?m −> signal‘ ;

/. <Expr>‘_get_instance_number (?m)‘
==> <Expr>‘?m −> instance‘ ;                                                    10

/. <Stmt>‘?var = _get_value (?m, ?e);‘
==> <Stmt>‘?var = ?m −> value ;‘ ;
```

C.1.8 Rule Set 8

```
start−timer is grammar cmod;

?dummy, ?taskid: Ident;
?timerid,?processid,?timerlength:Expr;
?newprocessid:Ident;
?tempprocid:Ident;

/. <Expr>‘start_timer(?timerid,?processid,?timerlength)‘
            & <Expr>‘?tempprocid‘ ~~ ?processid
            & ?newprocessid:= eval string−upcase ?tempprocid                    10
            & ?dummy := eval get−first−available−taskid |*task−ids*|
            & ?taskid := eval get−from−table−if−none−s ?tempprocid |*process−id−table*| ?dummy
            & ! ?taskid == ?dummy

==> <Expr>‘start_timer(?timerid,?newprocessid,?timerlength)‘;


/. <Expr>‘start_timer(?timerid,?processid,?timerlength)‘
            & <Expr>‘?tempprocid‘ ~~ ?processid
            & ?newprocessid:= eval string−upcase ?tempprocid                    20
            & ?dummy := eval get−first−available−taskid |*task−ids*|
            & ?taskid := eval get−from−table−if−none−s  ?tempprocid |*process−id−table*| ?dummy
            & ?taskid == ?dummy
            & test put−into−table−s ?procname ?taskid |*process−id−table*|
            & test update−taskid−list |*task−ids*|
==> <Expr>‘start_timer(?timerid,?newprocessid,?timerlength)‘;
```

C.1.9 Rule Set 9

```
stop−timer is grammar cmod;

?dummy, ?taskid: Ident;
?timerid,?processid:Expr;
?newprocessid:Ident;
?tempprocid:Ident;

/. <Expr>'stop_timer(?timerid,?processid)'
            & <Expr>'?tempprocid' ~~ ?processid
            & ?newprocessid:= eval string−upcase ?tempprocid                                    10
            & ?dummy := eval get−first−available−taskid |*task−ids*|
            & ?taskid := eval get−from−table−if−none−s ?tempprocid |*process−id−table*| ?dummy
            & ! ?taskid == ?dummy
==> <Expr>'stop_timer(?timerid,?newprocessid)';


/. <Expr>'stop_timer(?timerid,?processid)'
            & <Expr>'?tempprocid' ~~ ?processid
            & ?newprocessid:= eval string−upcase ?tempprocid
            & ?dummy := eval get−first−available−taskid |*task−ids*|                            20
            & ?taskid := eval get−from−table−if−none−s  ?tempprocid |*process−id−table*| ?dummy
            & ?taskid == ?dummy
            & test put−into−table−s ?procname ?taskid |*process−id−table*|
            & test update−taskid−list |*task−ids*|
==> <Expr>'stop_timer(?timerid,?newprocessid)';
```

Bibliography

[1] James M. Boyle. Abstract programming and program transformation - an approach to reusing programs. Technical report, Mathematics and Computer Science Division - Argonne National Laboratory, 1989.

[2] James M. Boyle. Practical transformation of functional programs for efficient execution: A case study. Technical report, Mathematics and Computer Science Division - Argonne National Laboratory, 1993.

[3] L.G.L.T. Meertens (Ed.). *Program Specification and Transformation.* North Holland Publishing Company, Amsterdam, 1987.

[4] Peter Pepper (Ed.). *Program Transformation and Programming Environments, NATO ASI Series, Vol. F8.* Springer-Verlag, Berlin, Heidelberg, New York, 1984.

[5] Martin S. Feather. A survey and classification of some program transformation approaches and techniques. Technical report, Information Sciences Institute, USC, 1987.

[6] Barbara Liskov. *Abstraction and Specification in Program Development.* MIT Press, Cambridge, 1986.

[7] Bernhard Möller, Helmut Partsch, and Steve Schuman (Eds.). *Lecture Notes in Computer Science: Formal Program Development.* Spring-Verlag, Berlin, Heidelberg, New York, 1993.

[8] Motorola. *Associated Control Procedure*, 1995.

[9] Robert Paige. Supercompilers - extended abstract. Technical report, Rutgers University, Department of Computer Science, 1984.

[10] Helmut A. Partsch. The cip transformation system. Technical report, Technical University of Munich, 1984.

[11] Helmut A. Partsch. *Specification And Transformation of Programs: A Formal Approach to Software Development.* Springer-Verlag, Berlin, Heidelberg, New York, 1990.

[12] Helmut A. Partsch. Formal problem specification on an algebraic basis. Technical report, Universtät Ulm Germany, 1993.

[13] Alberto Pettorossi and Maurizio Proietti. Rules and strategies for program transformation. Technical report, University of Rome Electronics Department, 1993.

[14] Andrew S. Tannenbaum. *Computer Networks.* Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[15] Thomas Weigert. Knowledge-based derivations of programs from specifications. Technical report, Motorola Inc., 1994.

[16] Patrick Henry Winston. *Artificial Intelligence.* Addison Wesley, Reading, MA, 1992.