rtScript: A Scripting Language for Writing Real-Time Programs in C

Youngtae Kim
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/cs_techreports
Part of the Programming Languages and Compilers Commons

Recommended Citation
http://lib.dr.iastate.edu/cs_techreports/65

This Article is brought to you for free and open access by Digital Repository @ Iowa State University. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Digital Repository @ Iowa State University. For more information, please contact hinefuku@iastate.edu.
rtScript: A Scripting Language for Writing Real-Time Programs in C

TR 92-22
Youngtae Kim

July 18, 1992

Iowa State University of Science and Technology
Department of Computer Science
226 Atanasoff
Ames, IA 50011
rtScript: A Scripting Language for Writing
Real-Time Programs in C

Youngtae Kim

TR 92-22

ABSTRACT

In hard real-time systems, tasks have to be performed with correct timing behavior as well as with correct functional behavior. The timing behavior is specified in terms of release times, deadlines, and permissible variances. rtScript is a scripting language used to specify timing constraints and periodic execution of tasks. Precedence and exclusion relationships between tasks are also specified in rtScript. The script translator translates programs written in rtScript into an intermediate language called rtScore, which serves as input to a static task scheduler. This paper describes the design and implementation of rtScript and the script translator.

July 18, 1992

Department of Computer Science
Iowa State University
1. Introduction

As technology develops, real-time applications are becoming more practical, and a number of new and complicated real-time applications are currently emerging. Examples of such applications are found in nuclear power plants, process control plants, and flight control systems. Also, many important applications, such as radar signal processing and airline reservation, have inherent real-time properties.

There are two types of real-time systems: soft real-time systems and hard real-time systems [1]. In soft real-time systems, tasks are performed as fast as possible, but may not be constrained to finish by specific times. On the other hand, in hard real-time systems, tasks have to be performed with correct timing behavior, as well as with correct functional behavior. Hard real-time computing does not simply mean fast execution of programs; it means programmers have the ability to specify the start and stop times for tasks. While fast execution is helpful to meet stringent timing requirements, fast execution alone does not guarantee timing predictability. The most important property of hard real-time systems is timing predictability, so it is a primary concern of hard real-time programmers to guarantee that applications complete within specified time bounds. In this paper, the phrase “real-time system” means hard real-time system.

Typically, tasks in real-time systems are characterized by timing constraints, and precedence and exclusion constraints. Timing constraint requires that the task executed once during each period of specified duration. If a task has a precedence constraint to another task, it has to finish before the other task starts within each period. If a task has an exclusion constraint to another task, the second task cannot execute during the time ranging from the start of the first task’s execution to
its finish within each period. Tasks in a real-time system are scheduled to satisfy these constraints. Usually, a number of different schedules satisfy these constraints. Task scheduling can be done in a static or dynamic fashion [1]. In static scheduling, scheduling is done before run time. A static scheduler uses the timing and constraint information for each task to determine the times during which the task will execute. A static scheduler requires knowledge of task execution constraints and resource requirements, such as release time, deadline, computation time and relationships with other tasks. In contrast, dynamic scheduling makes all scheduling decisions during program execution. Dynamic scheduling is more flexible than static scheduling because the dynamic scheduler’s decisions may depend on the current system state. If programmers cannot predetermine the behavior of the real-time system before execution of tasks, or if tasks are continually added or removed, dynamic scheduling is more useful. However, the costs of making dynamic scheduling decisions must be added to the real-time workload. A further problem with dynamic scheduling is that it is difficult for the real-time programmer to verify that all of the resources required for execution of each real-time task will be available when that task needs to execute.

A prescheduler is a program that schedules tasks in static fashion. This scheduler precalculates a scheduling table that specifies when each context switch should be performed. The scheduling table is finite in size. The run-time scheduler dispatches tasks according to the precomputed scheduling table. Upon completion of the tabulated schedule, the run-time scheduler starts over at the beginning of the table. Since the run-time cost of context switching is a small constant [2], prescheduled real-time systems offer the potential of higher performance than dynamic schedules. Though the problem of prescheduling tasks with real-time constraints is known to be NP-hard in general, there are effective scheduling algorithms and heuristics for prescheduling [3, 4].
2. System Overview

rtScript is a scripting language used to specify periodic execution of tasks. Real-time programmers use rtScript to define the timing behavior of tasks. This behavior is specified in terms of task release times, deadlines, and permissible variances. Precedence and exclusion constraints of tasks are also specified in rtScript. A script translator translates the program written in rtScript into the error-free input of a prescheduler called, in the figure below, the schedule editor. The rtScript language is designed in hierarchical structure so as to simplify programming. The relationship between the script translator and other components in the real-time programming environment is shown in Figure 2.1.

![Figure 2.1 Real-time programming environment](image-url)
In the schedule editor, tasks can be scheduled with timing information derived from program components and timing constraints specified by programmers before execution of the tasks. Additionally, the schedule editor performs link editing and certain link-time optimization.

Tasks in real-time systems can be separated into essential real-time tasks and optional real-time tasks. Such an organization is described in [5]. The time required to execute tasks in real-time systems varies greatly depending on the system’s workload, cache state, and bus connection. When precomputing schedules, the static scheduler guarantees the worst-case resource requirements to each essential task. In order to utilize the excess resources available when essential tasks do not require all of the resources allocated to them, the static scheduler prepares a contingency schedule in which optional tasks are allowed to execute when idle resources are available. Whenever essential tasks complete ahead of their deadlines, and CPU-time is available, the time may be allocated to optional tasks by the run-time scheduler. Optional tasks can also be prescheduled in the primary plan in addition to running on a contingency basis. Such optional tasks are specified in rtScript.

The script translator performs syntax checking, semantic consistency checking, and translation of scripts into an internal intermediate code to be processed by the scheduler editor. Additionally, the script translator invokes the rtC compiler (rtC is an acronym for real-time C) with appropriate parameterizations in order to translate particular C functions into task descriptions which consist of object code and the results of CPU-time analysis of the tasks. The schedule editor takes object code from the rtC compiler and timing specifications from the script translator as its input.
3. Survey of Related Research

A number of programming languages and language features for supporting various real-time programming environments have been implemented. Of the many real-time languages and mechanisms, only a small number of languages support reliable programming. Some of these are described briefly below.

Real-time Euclid

Real-time Euclid is a Pascal-like language designed specifically for straightforward schedulability analysis. To support timing analysis of tasks, the language contains no recursion and all loops execute a constant number of iterations. To synchronize processes, Real-time Euclid uses various dynamic mechanisms: monitors, waits, and broadcasts [6]. Real-time Euclid supports a model where processes contend dynamically for access to shared resource, and the worst-case amount of time waiting for each shared resource is taken into account in computing the execution time. Our research is different from the research on Real-time Euclid in that rtC is more expressive than Real-time Euclid, and we synchronize access to shared resources using static precedence and exclusion constraints rather than dynamic monitor locking.

Spring-C

Similar to our research, Spring system uses a scripting language called SDL (System Description Language) to describe periodic execution of tasks written in Spring-C [7]. In the Spring system, most timing constraints are specified in Spring-C rather than in SDL. Analysis of task execution times is done at the intermediate code level using instruction costs determined experimentally. Because of the unconstrained expressive power of SDL, some SDL/Spring-C programs cannot be analyzed for schedulability. Currently, Spring-C’s designer recommends that programmers simply attempt to compile their programs to see if they are analyzable. In contrast, rtScript is analyzable because rtScript is deterministic, i.e., rtScript has no conditional constructs. We consider Spring-C’s lack of clear guidelines to characterize legal programs as a significant design problem,
especially if Spring-C programs might be generated automatically by other software tools.

**Flex language**

J. W. S. Liu and other researchers provide imprecise computation techniques for scheduling flexibility in the Flex language [5]. They decompose every real-time task into a mandatory subtask and an optional subtask. A mandatory subtask executes with timing constraints, but gives only an approximation of the desired result, not guaranteeing the precise result. An optional subtask is used to improve upon the approximate result computed by the mandatory subtask. The optional subtask becomes ready for execution when the mandatory subtask is completed. The system can terminate the optional subtask when the available CPU-time is exhausted. The imprecise computation technique avoids timing faults, and gives an approximate result with acceptable quality even when the system cannot produce the exact result within a given time. Although the concept of optional subtask is similar to our notion of optional task, our optional tasks are independent tasks not necessarily related to particular essential tasks. To support imprecise computation in Flex, automatic timing measurement of program execution times is introduced in the Flex environment [8]. This measurement uses statistical analysis to determine execution times empirically. One of Flex’s innovations is the notion of *Performance polymorphism*. Multiple algorithms are executed in parallel for computation of a given function. These algorithms compute the same result with varying degrees of precision and different computation requirements. Among the results produced within the available time, the system selects the value produced by the most precise algorithm.
4. Description of rtScript

The syntactic and semantic design of rtScript is discussed in this section.

4.1 Language Overview

The primary objective of rtScript programs is to specify the real-time behavior of a set of tasks. In rtScript, tasks are syntactically grouped together within families and processes. Each task has a release time and a deadline which are specified by a programmer, and each task is started exactly once within the period of the process or family it belongs to. Release times and deadlines are specified relative to the enclosing scope. Precedence and exclusion relationships between tasks are specified in rtScript. If no relationships are specified, tasks may preempt each other at run time.

In rtScript, a group of cooperating tasks and families which share the same period is defined in a process block. Timing constraints of tasks and relationships between them are specified in a process block. A family block is used to group cooperating periodic tasks and subfamilies which share the same period. The structure of a family block is almost the same as the structure of a process block. However, while a family block can appear in another family block or process block, no nesting of process blocks is permitted.

Since most optional tasks make use of language features for which the rtC compiler cannot derive upper bounds on execution times, rtScript programmers specify how much time should be allotted to the optional task during each period. There are two kinds of optional tasks in rtScript: soft tasks and continuation tasks. Soft tasks are restarted during each period. A continuation task continues execution wherever it left off at the end of its last CPU-time allotment.

Tasks can share global variables within a particular process, but tasks in different processes cannot share global variables with each other. If multiple processes refer to the same global variables, the global variables are instantiated independently for each process that accesses those global variables.
4.2 Structure of rtScript

rtScript is a C-like language. C-preprocessor directives, such as `#include` and `#define` statements, can be used, and C-style comments are used in rtScript programs. The rtScript program consists of one or more process statements. Each process statement contains one or more task or family statements. A family statement can contain one or more task statements or family statements. The language definition places no bound on the number of levels of nested family definitions. Precedence and exclusion relationships are specified in process and family statements. The hierarchical structure of rtScript is shown in Figure 4.2.1.

![Hierarchical structure of the scripting program](image)

Figure 4.2.1 Hierarchical structure of the scripting program
4.2.1 rtC Source File Declaration

After reading an input program, the script translator invokes the rtC compiler to compile the rtC source files that contain the relevant task definitions. These files are identified to the script translator using special source-file declarations which appear prior to all process declarations in the rtScript code. Their syntax follows.

Syntax:

```
source "source_file_name";
```

Example:

```
source "DiskManager.c";
```

The rtC compiler for the source file `DiskManager.c` is invoked by the script translator, producing the object code `DiskManager.o` and the intermediate file `DiskManager.i`. A `source` statement must precede the process statement that makes reference to the C functions defined within the source file. The script translator gets information about the times required to execute the tasks defined in particular rtC source files by examining the intermediate .i files output by the rtC compiler. In the above example, `DiskManager.i` contains the following:

```
Server$bound = 0.01
ArmController$bound = 0.005
SectorBuffer$bound = 0.004
Tracking$unbound
$$
```

Server, ArmController, and SectorBuffer are time-bounded, and have the computation times 0.01 sec, 0.005 sec, and 0.004 sec respectively. The reason that Tracking does not have a bound is because Tracking makes use of language features that cannot be analyzed. This is acceptable because Tracking is an optional task in the rtScript program. While essential rtScript tasks must be time-bounded, optional rtScript tasks may be time-bounded or time-unbounded.
4.2.2 Time Units

In rtScript, time is expressed in terms of nsec, µsec, msec, sec, or min. All timing specifications consist of a real number accompanied by an optional time unit keyword chosen from the following: nanosec, microsec, msec, sec, min. If the time units are not specified, the script translator assumes the default time unit, which is initially msec. The default time unit can be redefined using the following syntax:

Syntax:

\[ \text{default	extunderscore time	extunderscore unit } \text{time	extunderscore unit}; \]

Example:

\[ \text{default	extunderscore time	extunderscore unit microsec;} \]

The scope of a default-time-unit declaration ranges from the point of the declaration to a subsequent default-time-unit redeclaration or to the end of the file, whichever comes first.

4.2.3 Process Statements

Within a process statement, task declarations, family declarations and timing relationships are specified. Each process has a period. When specifying the periods of processes, programmers may describe either exact periods or periods with ranges of acceptable variation if the variations are allowed. A process statement begins with the process keyword. Each process statement has a period and an optional variance. Omitting the variance is equivalent to specifying a variance of 0.
Syntax:

```
process process_name(period, variance) {
    ....
}
```

or

```
process process_name(period) {
    ....
}
```

Example:

```
process DiskController(40 msec, 10 msec) {
    ....
}
```

The DiskController process has a period of 40 msec and a variance of 10 msec. The allowed period is the specified period plus or minus the variance, so the DiskController process has a period between 30 and 50 msec.

4.2.4 Family Statements

The purpose of a family declaration is to group together a collection of tasks and subfamilies, all of which have the same period of execution. The family’s period is determined by dividing the period of the enclosing process or family by the value of the family’s occurrences parameter. If an optional slack is specified in the header of the family declaration, then individual periods of the family are allowed to vary from the period calculated above by the specified slack. If a slack is not given, no variation is allowed.
Syntax:

```plaintext
family family_name(occurrences, slack) {  
    ....
}  
or
family family_name(occurrences) {  
    ....
}
```

Example:

```plaintext
process DiskController(40,10) {  
    family Scheduler(4,5) {  
        ....
    }
}
```

In this example, the Scheduler family executes four times during each DiskController period. Assuming the default time unit is msec, the Scheduler family has p/4 msec as its period where p is the period of the DiskController process. The time allotted to each occurrence of the Scheduler family may vary from the period by 5 msec. For instance, the periods of the four occurrences of the Scheduler family might be 5 msec, 10 msec, 15 msec and 10 msec.

### 4.2.5 Task Statements

Every rtScript task declaration refers to a rtC function by the same name. The rtC functions are defined in rtC source files identified with rtScript source directives. The computation time of the task is predetermined analytically whenever possible by the rtC compiler [2]. If a task has a release time r and a deadline d, the computation time of the task must be less than (d − r).

Syntax:

```
task task_name(release_time, deadline), task_name(release_time, deadline), ...;
```
Suppose that a task **ArmController** has a release time of 5 msec and a deadline of 35 msec, and that a task **SectorBuffer** has a release time of 4 msec and a deadline of 35.5 msec. Assuming the default time unit is msec, these tasks are described as follows:

```plaintext
task ArmController(5, 35);
task SectorBuffer(4, 35.5);
```

or

```plaintext
task ArmController(5, 35), SectorBuffer(4, 35.5);
```

Declarations of optional tasks specify not only release times and deadlines, but also allotment times and allowed variations of the allotment times. The allotment time is the amount of time the task is allowed to run during each period.

Syntax:

```plaintext
soft_task task_name(release_time, deadline, allotment, variance);
cont_task task_name(release_time, deadline, allotment, variance);
```

Examples:

```plaintext
soft_task Tracking(5, 35, 3, 2);
cont_task Optimizer(0, 1.5, 1.2, 1.2);
```

Assuming the default time unit is msec, the soft task **Tracking** has 5 msec, 35 msec, 3 msec and 2 msec as its release time, deadline, allotment and variance respectively. And the continuation task **Optimizer** has 0 msec, 1.5 msec, 1.2 msec and 1.2 msec as its release time, deadline, allotment and variance respectively. Note that, since **Optimizer**’s variance equals its allotment, execution of the **Optimizer** task is completely optional.
4.2.6 Task Relationships

Programmers may require that certain tasks precede execution of other tasks or that particular tasks exclude execution of other tasks. If a task \( t_1 \) precedes \( t_2 \), \( t_1 \) has to finish before \( t_2 \) starts within each period. If a task \( t_1 \) excludes \( t_2 \), once \( t_1 \) has started its computation, it cannot be pre-empted by \( t_2 \) until \( t_1 \) completes its execution for a given period. Task relationships are specified within the innermost process or family statement that encloses the relevant task declarations.

Syntax:

\[
\text{task}_\text{name} \text{ precedes } \text{task}_\text{name}; \\
\text{task}_\text{name} \text{ excludes } \text{task}_\text{name};
\]

Example:

\[
\text{process DiskController}(40, 10) \{ \\
\text{task ArmController}(5, 35), \text{SectorBuffer}(4, 35.5); \\
\text{soft_task Tracking}(5, 35, 3, 2); \\
\quad \text{ArmController precedes Tracking}; \\
\quad \text{ArmController excludes SectorBuffer}; \\
\}
\]

In the above example, the ArmController task precedes the Tracking task during each period of the DiskController process. The SectorBuffer task is not allowed to run concurrently with the ArmController task.

When precedence or exclusion relationships refer to tasks declared within inner-nested families, the occurrence number of the enclosing family is used to uniquely identify the tasks. Family occurrences are numbered in increasing order from 0. Consider the following rtScript code.
family Scheduler(4, 5) {
    family SeekScheduler(2, 1) {
        task Seek(0, 1.5);
    }
    task FindNext(1, 8);

    SeekScheduler[2].Seek precedes FindNext;
}

In the above example, the Seek task in the third occurrence of the SeekScheduler family must terminate before the FindNext task begins to execute. Note that families may nest within families, as in the following example.

process DiskController(40, 10) {
    task Server(5, 40);
    family Scheduler(4, 5) {
        family SeekScheduler(2, 1) {
            task Seek(0, 1.5);
        }
    }

    Server precedes Scheduler[3].SeekScheduler[1].Seek;
}

In this example, the Server task precedes execution of the Seek task in the second occurrence of the SeekScheduler family within the fourth occurrence of the Scheduler family.

4.3 Using rtScript

In order to use rtScript, a programmer must create the program in a file whose name ends in .rts such as DiskManager.rts, then compile it with the command:

```
rtsc DiskManager.rts
```

One or more files can be specified as input on the rtsc command line. Assuming no errors in its input, the compilation proceeds silently, producing output files which end .score. Thus, if DiskManager.rts has no errors, the output is DiskManager.score. This output serves as input to the schedule editor.
5. Overview of Implementation

This section gives an overview of the script translator’s implementation.

5.1 The Parser and Lexical analyzer

The script translator’s scanner is generated by flex, and its parser is generated by bison. The flex input file is scan.l, which is 275 lines long, and the bison input file is gram.y, which consists of 652 lines. bison and flex are used instead of yacc and lex because they are compatible with gnu C++.

5.2 The Script Translator

The script translator is written in C++. There are 4 major classes in the source program:

- word.hh, word.cc - a permanent word manager, i.e., a storage of strings (69 lines)
- sym.hh sym.cc - a symbol table management routine (680 lines)
- getdata.hh, getdata.cc - a communicator with a real-time C compiler (126 lines)
- error.hh, error.cc - an error message manager (76 lines)

All of the source code is listed in the appendix.
6. Discussion of the Output of rtScript

The output of rtScript is discussed in this section.

6.1 Structure of the Output

The main responsibility of the script translator is to change the hierarchical rtScript input into a flat, error-free description of tasks called rtScore. rtScore is the intermediate scripting language which serves as input to the scheduler editor. rtScore consists of one or more process blocks corresponding to process blocks defined in the rtScript program. Each process block specifies the timing constraints and task relationships between the tasks declared within the process. The process block consists of two basic components: a timing constraint component and a task relationship component. The process block has the following structure.

\[
\text{process } \text{process\_name} \\
\text{# timing constraints} \\
\text{....} \\
\text{# variables} \\
\text{....} \\
\text{# task relations} \\
\text{....}
\]

In rtScore, comments are introduced by the \# symbol. All of the text to the right of the \# symbol on the same line is treated as a comment. In the structure above, \# timing constraints is the comment to introduce the timing constraint component which specifies the timing behavior of tasks. Certain timing constraints are described in terms of symbolic constants. The variables section of a rtScore process block, which is introduced by a \# variables comment, describes the ranges of values that particular constants are allowed to represent. The task relations component, which begins with the \# task relations comment, describes task precedence and exclusion constraints. All times are expressed in terms of seconds.

For example, the following rtScript program:
process DiskController(40 msec, 10 msec) {
    ....
}

is translated into:

process DiskController
# timing constraints
    ....
    # variables
    DiskController$(0.03:0.05)
    ....
    # task relations
      ....

The period of a process is represented by the name of the process catenated with a $ sign. DiskController$(0.03:0.05) in the variables section signifies that the DiskController process has a period of 40±10 msec (i.e. the period is between 30 and 50 msec).

6.2 Timing Constraints

In the timing constraint part, a computation time, release time and deadline is specified for each task defined in the source program. Each line of rtScore supplies the timing information for a different task. The syntax for periodic tasks, soft tasks, and continuation tasks is illustrated below:

    task_name(object_file_name,computation_time,release_time:deadline)
    soft task_name(object_file_name,allotment%variance,release_time:deadline)
    cont task_name(object_file_name,allotment%variance,release_time:deadline)

In each of these cases, object_file_name represents the full path of the object file that contains the task’s translation, and task_name is the name of the C module contained within the object file.

For example, the following two rtScript lines:

    task  ArmController(5, 35);
    soft_task Tracking(5, 35, 3, 2);

are translated into:
ArmController(/usr/rtc/DiskManager.o,0.004,0.005:0.035)
soft Tracking(/usr/rtc/DiskManager.o,0.003%0.002,0.005:0.035)

/usr/rtc/DiskManager.o is the object file that contains the translations of the rtC functions ArmController and Tracking. The computation time of the task ArmController is 0.004 sec, and 0.005:0.035 denotes the task’s release time of 0.005 sec and its deadline of 0.035 sec relative to the start of its period. The allotment and variance of the optional Tracking task is expressed as 0.003%0.002.

Tasks within families are named in the rtScore file using a concatenation of the family name, a $ sign, the task’s instantiation number within the family, and the task name. For example, if task t defined within family f executes once during each of f’s three occurrences, then the rtScore file must describe the real-time behavior of each of the three instantiations of t. These instantiations are named f$0t, f$1t, and f$2t. The names of families nested within families are similarly qualified. Suppose, for example, that family p executes twice within each period of family q. Then, the two instantiations of family p are named q$0p and q$1p. Tasks within p are named by appending the appropriate suffix to either of the names that refer to instantiations of p. Here are a couple of examples.

Example1:

The input program:

```plaintext
process DiskController(40,10) {
    family Scheduler(4,5) {
        task FindNext(1,8);
    }
}
```

is translated into:
process DiskController

# timing constraints

Scheduler$0FindNext(/usr/rtc/Scheduler.o,0.002,Scheduler$0+0.001:Scheduler$0+0.008)
Scheduler$1FindNext(/usr/rtc/Scheduler.o,0.002,Scheduler$1+0.001:Scheduler$1+0.008)
Scheduler$2FindNext(/usr/rtc/Scheduler.o,0.002,Scheduler$2+0.001:Scheduler$2+0.008)
Scheduler$3FindNext(/usr/rtc/Scheduler.o,0.002,Scheduler$3+0.001:Scheduler$3+0.008)

# variables

DiskController$(0.03:0.05)
Scheduler$0(0:0)
Scheduler$1(Scheduler$0+DiskController$/4−0.005:Scheduler$0+DiskController$/4+0.005)
Scheduler$2(Scheduler$1+DiskController$/4−0.005:Scheduler$1+DiskController$/4+0.005)
Scheduler$3(Scheduler$2+DiskController$/4−0.005:Scheduler$2+DiskController$/4+0.005)
Scheduler$3(DiskController$−DiskController$/4−0.005:DiskController$−DiskController$/4+0.005)

# task relations

The object code for FindNext is found in the file /usr/rtc/Scheduler.o. The upper bound on the time required to execute FindNext is 0.002 sec. The release time and the deadline of FindNext are relative to the time at which the corresponding family begins to execute. The beginning time of each family instantiation is represented by the name of the family concatenated with a $ sign and the instance number.

In the variables block, rtScore defines constraints on the symbols used in specifying timing constraints of tasks. Since slack is allowed in the periods of individual family occurrences, the beginning time of a family occurrence is specified in terms of the beginning time of the previous instance of the family. For the first occurrence of a family, the beginning time is constrained in terms of the beginning time of the enclosing family or process. The beginning time of a process is 0. For the last occurrence of a family, the beginning time depends on not only the beginning time of the preceding occurrence, but also on the period of the enclosing family or process. In the
above, the range of Scheduler\$n is constrained as follows:

if n = 0 :
    Scheduler\$n = 0
if n > 0 :
    Scheduler$\!(n−1) + period − slack ≤ Scheduler\$n ≤ Scheduler$\!(n−1) + period + slack
    if Scheduler\$n is the last occurrence :
        enclosing_period − (period + slack) ≤ Scheduler\$n ≤ enclosing_period − (period − slack)

In the above expressions, period represents the derived period of the Scheduler family, and enclosing_period represents the period of the enclosing process, DiskController. The period and slack of the Scheduler family are DiskController$/4 sec, and 0.005 sec respectively, so the range of Scheduler\$0 is (0:0), the range of Scheduler\$1 is (Scheduler\$0+DiskController$/4−0.005:Scheduler\$0+DiskController$/4+0.005), and so on. Since Scheduler\$3 is the last occurrence, its range is also (DiskController$–DiskController$/4−0.005:DiskController$–DiskController$/4+0.005).

Example2:

The input program:

```
process DiskController(40,10) {
    family Scheduler(4,5) {
        family SeekScheduler(2,1) {
            task Seek(0,1.5);
        }
    }
}
```

is translated into:
process DiskController
# timing constraints
  Scheduler$0SeekScheduler$0Seek(/usr/rtc/Scheduler.o,0.001,Scheduler$0SeekScheduler$0+0.0015)
  Scheduler$0SeekScheduler$1Seek(/usr/rtc/Scheduler.o,0.001,Scheduler$0SeekScheduler$1+0.0015)
  Scheduler$1SeekScheduler$0Seek(/usr/rtc/Scheduler.o,0.001,Scheduler$1SeekScheduler$0+0.0015)
  Scheduler$1SeekScheduler$1Seek(/usr/rtc/Scheduler.o,0.001,Scheduler$1SeekScheduler$1+0.0015)
  Scheduler$2SeekScheduler$0Seek(/usr/rtc/Scheduler.o,0.001,Scheduler$2SeekScheduler$0+0.0015)
  Scheduler$2SeekScheduler$1Seek(/usr/rtc/Scheduler.o,0.001,Scheduler$2SeekScheduler$1+0.0015)
  Scheduler$3SeekScheduler$0Seek(/usr/rtc/Scheduler.o,0.001,Scheduler$3SeekScheduler$0+0.0015)
  Scheduler$3SeekScheduler$1Seek(/usr/rtc/Scheduler.o,0.001,Scheduler$3SeekScheduler$1+0.0015)

# variables
  DiskController$(0.03:0.05)
  Scheduler$0(0:0)
  Scheduler$0SeekScheduler$0(Scheduler$0:Scheduler$0)
  Scheduler$0SeekScheduler$1(Scheduler$0SeekScheduler$0+(Scheduler$1−Scheduler$0)/2−0.001:Scheduler$0SeekScheduler$0+(Scheduler$1−Scheduler$0)/2+0.001)
  Scheduler$0SeekScheduler$1(Scheduler$1−(Scheduler$1−Scheduler$0)/2−0.001:Scheduler$1SeekScheduler$0+(Scheduler$1−Scheduler$0)/2+0.001)
  Scheduler$1SeekScheduler$0(Scheduler$1:Scheduler$1)
  Scheduler$1SeekScheduler$1(Scheduler$1SeekScheduler$0+(Scheduler$2−Scheduler$1)/2−0.001:Scheduler$1SeekScheduler$0+(Scheduler$2−Scheduler$1)/2+0.001)
  Scheduler$1SeekScheduler$1(Scheduler$2−(Scheduler$2−Scheduler$1)/2−0.001:Scheduler$2−Scheduler$1)/2+0.001)
  Scheduler$2(Scheduler$1+DiskController$/4−0.005:Scheduler$1+DiskController$/4+0.005)
  Scheduler$2SeekScheduler$0(Scheduler$2:Scheduler$2)
  Scheduler$2SeekScheduler$1(Scheduler$2SeekScheduler$0+(Scheduler$3−Scheduler$2)/2−0.001:Scheduler$2SeekScheduler$0+(Scheduler$3−Scheduler$2)/2+0.001)
  Scheduler$2SeekScheduler$1(Scheduler$3−(Scheduler$3−Scheduler$2)/2−0.001:Scheduler$3−(Scheduler$3−Scheduler$2)/2+0.001)
Scheduler$3(Scheduler$2+DiskController$/4−0.005:Scheduler$2+DiskController$/4+0.005)
Scheduler$3(DiskController$−DiskController$/4−0.005:DiskController$−DiskController$/4+0.005)
Scheduler$3SeekScheduler$0(Scheduler$3:Scheduler$3)
Scheduler$3SeekScheduler$1(Scheduler$3SeekScheduler$0+(DiskController$−Scheduler$3)/2−0.001:Scheduler$3SeekScheduler$0+(DiskController$−Scheduler$3)/2+0.001)
Scheduler$3SeekScheduler$1(DiskController$−(DiskController$−Scheduler$3)/2−0.001:DiskController$−(DiskController$−Scheduler$3)/2+0.001)

# task relations

The Seek task is in the file /usr/rtc/Scheduler.o, and each task has the computation time 0.001 sec. The release time and the deadline are $t + 0$ sec and $t + 0.015$ sec respectively, where $t$ is the beginning time of each SeekScheduler family occurrence.

The period and the slack of the family SeekScheduler are $p/2$ sec and 0.001 sec respectively where $p$ is the period of the Scheduler enclosing family, so the range of Scheduler$0SeekScheduler$0 is (0:0), the range of Scheduler$0SeekScheduler$1 is (Scheduler$0SeekScheduler$0+(Scheduler$1−Scheduler$0)/2−0.001:Scheduler$0SeekScheduler$0+(Scheduler$1−Scheduler$0)/2+0.001) and (Scheduler$1−(Scheduler$1−Scheduler$0)/2−0.001:Scheduler$1−(Scheduler$1−Scheduler$0)/2+0.001), and so on. Constraints on the values of Scheduler$0$, Scheduler$1$, Scheduler$2$, Scheduler$3$ are as described above.

6.3 Timing Relationships

In rtScore, a precedence relationship is expressed with the symbol ‘;’, and an exclusion relationship is expressed with the symbol ‘!’ For example, the rtScript relationships:

Server precedes ArmController;
ArmController excludes SectorBuffer;

are translated into:
If the above relationships describe tasks defined within a family, then their translation results in specific constraints relating all relevant pairs of task instantiations. Consider, for example, the following program:

```plaintext
process DiskController(40,10) {
    task Server(5,40);
    family Scheduler(4,5) {
        family SeekScheduler(2,1) {
            task Seek(0,1.5);
            cont_task Optimizer(0,1.5,1.2,1.2);
            Seek precedes Optimizer;
        }
        task FindNext(1,8);
        SeekScheduler[1].Seek precedes FindNext;
    }
    Scheduler[0].FindNext precedes Server;
}
```

This is translated into:

```
# task relations
(Scheduler$0FindNext;Server)
(Scheduler$0SeekScheduler$1Seek;Scheduler$0FindNext)
(Scheduler$1SeekScheduler$1Seek;Scheduler$1FindNext)
(Scheduler$2SeekScheduler$1Seek;Scheduler$2FindNext)
(Scheduler$3SeekScheduler$1Seek;Scheduler$3FindNext)
(Scheduler$0SeekScheduler$0Seek;Scheduler$0SeekScheduler$0Optimizer)
(Scheduler$0SeekScheduler$1Seek;Scheduler$0SeekScheduler$1Optimizer)
(Scheduler$1SeekScheduler$0Seek;Scheduler$1SeekScheduler$0Optimizer)
(Scheduler$1SeekScheduler$1Seek;Scheduler$1SeekScheduler$1Optimizer)
(Scheduler$2SeekScheduler$0Seek;Scheduler$2SeekScheduler$0Optimizer)
(Scheduler$2SeekScheduler$1Seek;Scheduler$2SeekScheduler$1Optimizer)
(Scheduler$3SeekScheduler$0Seek;Scheduler$3SeekScheduler$0Optimizer)
(Scheduler$3SeekScheduler$1Seek;Scheduler$3SeekScheduler$1Optimizer)
```

Since there is only one occurrence of `Scheduler[0].FindNext`, there is only one constraint related to this task in the rtScore code. On the other hand, there are 4 different occurrences of `SeekScheduler[1]`, so this leads to 4 different restrictions in the rtScore translation. Similarly,
there are 8 instances of Seek and Optimizer, so there are 8 relationships between Seek and Optimizer tasks in the rtScore code.
7. Further Research

rtScript is designed to describe timing behaviors of real-time systems clearly and concisely. Whether we have achieved this goal remains to be seen. Experimental use of rtScript to implement interesting real-time applications will help evaluate the degree to which we have succeeded in our design goals. Theoretical studies characterizing the expressive limitations of rtScript would also be helpful.

To reduce the likelihood that rigid timing constraints preclude scheduling, rtScript allows programmers to specify tolerances for most timing constraints. Since most existing scheduling algorithms and heuristics use exact times for scheduling constraints, we need to develop and evaluate new scheduling techniques that take advantage of the additional flexibility afforded to find optimal constraint values within the allowed tolerances.

In the design of our real-time programming environment, a primary issue is the design of a prescheduler that analyzes the behaviors of periodic tasks and constructs static schedules for the tasks. We intend to survey the scheduling algorithms and heuristics to be used in the schedule editor. rtScore is an intermediate notation for communication between the script translator and the schedule editor. The current design of rtScore is motivated primarily by implementation issues related to the script translator. As work proceeds on our schedule editor implementation, we anticipate that a variety of changes to the rtScore notation will be desirable. Another issue to be addressed in the context of our schedule editor implementation is whether rtScript is sufficiently restrictive so as to facilitate straightforward schedulability analysis and automatic construction of schedule tables.

To support efficient resource utilization, the schedule editor prepares contingency plans to accompany the schedules of essential real-time tasks. Issues such as fairness and priorities need to be addressed in the context of optional computations, and techniques must be developed to implement solutions to these problems.
8. Acknowledgement

I would like to thank my major professor, Dr. Kelvin Nilsen for providing significant help during the design and development of this work. I would also like to thank the members of my POS committee, Dr. Gary Leavens and Dr. Charles Wright for their advice and encouragement.
REFERENCES


Appendix A: Example rtScript Program

Input:

source "DiskManager.c";
source "Scheduler.c";

default_unit msec;

process DiskController(40,10) {

    /*
    ** declaration of tasks
    */
    task Server(5,40);
    task ArmController(5,35), SectorBuffer(4,35.5);
    soft_task Tracking(5,35,3,2);

    /*
    ** declaration of family
    */
    family Scheduler(4,5) {
        family SeekScheduler(2,1) {
            task Seek(0,1.5);
            cont_task Optimizer(0,1.5,1.2,1.2);
            Seek precedes Optimizer;
        }
        task FindNext(1,8);
        SeekScheduler[1].Seek precedes FindNext;
    }

    /*
    ** task relationships
    */
    Server precedes ArmController;
    Server precedes SectorBuffer;
    ArmController excludes SectorBuffer;
    ArmController precedes Tracking;
    Scheduler[0].FindNext precedes Server;
}
process DiskController
# timing constraints
Server:(/usr/rtc/DiskManager.o,0.01,0.005:0.04)
ArmController:(/usr/rtc/DiskManager.o,0.005,0.005:0.035)
SectorBuffer:(/usr/rtc/DiskManager.o,0.004,0.004:0.035)
soft Tracking:(/usr/rtc/DiskManager.o,0.005,0.005:0.035)
Scheduler$0FindNext(/usr/rtc/Scheduler.o,0.002,Scheduler$0+0.001:Scheduler$0+0.008)
Scheduler$0SeekScheduler$0Seek(/usr/rtc/Scheduler.o,0.001,Scheduler$0SeekScheduler$0+0.001:Scheduler$0SeekScheduler$0+0.008)
Scheduler$0SeekScheduler$0Optimizier(/usr/rtc/Scheduler.o,0.001,Scheduler$0SeekScheduler$0+0.001:Scheduler$0SeekScheduler$0+0.008)
Scheduler$1FindNext(/usr/rtc/Scheduler.o,0.002,Scheduler$1+0.001:Scheduler$1+0.008)
Scheduler$1SeekScheduler$1Seek(/usr/rtc/Scheduler.o,0.001,Scheduler$1SeekScheduler$1+0.001:Scheduler$1SeekScheduler$1+0.008)
Scheduler$1SeekScheduler$1Optimizier(/usr/rtc/Scheduler.o,0.001,Scheduler$1SeekScheduler$1+0.001:Scheduler$1SeekScheduler$1+0.008)
Scheduler$2FindNext(/usr/rtc/Scheduler.o,0.002,Scheduler$2+0.001:Scheduler$2+0.008)
Scheduler$2SeekScheduler$2Seek(/usr/rtc/Scheduler.o,0.001,Scheduler$2SeekScheduler$2+0.001:Scheduler$2SeekScheduler$2+0.008)
Scheduler$2SeekScheduler$2Optimizier(/usr/rtc/Scheduler.o,0.001,Scheduler$2SeekScheduler$2+0.001:Scheduler$2SeekScheduler$2+0.008)
Scheduler$3FindNext(/usr/rtc/Scheduler.o,0.002,Scheduler$3+0.001:Scheduler$3+0.008)
Scheduler$3SeekScheduler$3Seek(/usr/rtc/Scheduler.o,0.001,Scheduler$3SeekScheduler$3+0.001:Scheduler$3SeekScheduler$3+0.008)
Scheduler$3SeekScheduler$3Optimizier(/usr/rtc/Scheduler.o,0.001,Scheduler$3SeekScheduler$3+0.001:Scheduler$3SeekScheduler$3+0.008)

# variables
DiskControllers(0.03:0.05)
Scheduler$0(0:0)
Scheduler$0SeekScheduler$0(Scheduler$0:Scheduler$0)
Scheduler$1(Scheduler$0:Scheduler$0)
Scheduler$2(Scheduler$1:Scheduler$1)
Scheduler$3(Scheduler$2:Scheduler$2)

# task relations
(Server;ArmController)
(Server;SectorBuffer)
(ArmController;SectorBuffer)
(ArmController;Tracking)
(Scheduler$0FindNext;Server)
(Scheduler$0SeekScheduler$0FindNext)
(Scheduler$1(Scheduler$0:Scheduler$0)
(Scheduler$2(Scheduler$1:Scheduler$1)
(Scheduler$3(Scheduler$2:Scheduler$2)

# task constraints
(Server;ArmController)
(Server;SectorBuffer)
(ArmController;SectorBuffer)
(ArmController;Tracking)
(Scheduler$0FindNext;Server)
(Scheduler$0SeekScheduler$0FindNext)
(Scheduler$1(Scheduler$0:Scheduler$0)
(Scheduler$2(Scheduler$1:Scheduler$1)
(Scheduler$3(Scheduler$2:Scheduler$2)

A-2
Appendix B: Source Code of the rtScript Translator

(gram.y)

/*
   Parser for Script Translator
*/

/* type of identifier */
%token TK_IDENTIFIER

/* source file name */
%token TK_FILENAME

/* type of value */
%token TK_INT TK_DOUBLE

/* time units */
%token TK_MIN TK_SEC TK_MSEC TK_MICROSEC TK_NANOSEC

/* keywords */
%token TK_SOURCE TK_DEFAULT_UNIT
%token TK_PROCESS TK_FAMILY TK_TASK TK_SOFT_TASK TK_CONT_TASK
%token TK_PRECEDES TK_EXCLUDES

%
#include <stream.h>
#include <string.h>
#include <ctype.h>
#include "error.hh"
#include "word.hh"
#include "sym.hh"
#include "getdata.hh"

#define yyerror(s) errmsg.error(s)
extern ErrorMessage errmsg;
extern WordManager words;
SymTable symtable;
GetData getdata;
Stack stack;
extern ostream *fout;

struct time_rel { // list of families for task variable
   sym *p;
   char s[BufSize];
} ;

static char msgbuf[BufSize];
static categories task_option;
double default_time_unit = 0.001;

boolean error_occurred = false;
static boolean
    process_decl_err = false,
    family_decl_err = false,
    task_rel_err = false;

static int family_nest = 0;

int yylex();
%

/* the attribute stack entry */
%union {
    char    *s;
    int      i;
    double   d;
    struct sym    *p;
    struct time_rel  *t;
}

%start script       /* start symbol is script */
%

script : external_declarations
    | ;

external_declarations : external_declaration
                      | external_declarations external_declaration
                      ;

external_declaration : source_file
                      | process_definition
                      | error ';
                      {
                          error_occurred = true;
                          sprintf(msgbuf,"misuse keyword");
                          errmsg.error(msgbuf);
                      }
                      ;

source_file : TK_SOURCE TK_FILENAME ';
             {
                 getdata.execute($<s>3);
             }
| TK_SOURCE error ';
             {
                 error_occurred = true;
                 sprintf(msgbuf,"source file declaration error");
                 errmsg.error(msgbuf);
             }
process_definition
  :default_unit process_head process_body
    {
      if (!error_occurred)
        symtable.pr_process(stack.pop());

      process_decl_err = false;
      symtable.remove_all();
    }

default_unit
  :TK_DEFAULT_UNIT time_unit ';'
    {
      default_time_unit = $<d>2;
    }
  |TK_DEFAULT_UNIT error ';'
    {
      error_occurred = true;
      sprintf(msgbuf,"default_time error");
      errmsg.error(msgbuf);
    }
  | /* nothing */
    ;

time_unit
  :TK_MIN
    {
      $<d>$ = 60.0;
    }
  |TK_SEC
    {
      $<d>$ = 1.0;
    }
  |TK_MSEC
    {
      $<d>$ = 0.001;
    }
  |TK_MICROSEC
    {
      $<d>$ = 0.000001
    }
  |TK_NANOSEC
    {
      $<d>$ = 0.000000001;
    }
    ;

process_head
  :TK_PROCESS process_name '(' period ',' variance')'
    {
      sym *sp;
      if (sp = symtable.lookup($<s>2, 0)) 

error_occurred = process_decl_err = true;
sprintf(msgbuf,"redeclaration of process: %s", sp->name);
errmsg.error(msgbuf);
} else if ($<d>4 < $<d>6) {
    error_occurred = process_decl_err = true;
    sprintf(msgbuf,"time error in process %s", $<s>2);
    errmsg.error(msgbuf);
} else
    stack.push(symtable.add_process(stack.top_of(),
        $<s>2, $<d>4, $<d>6));

} [TK_PROCESS process_name ' (' period ') ']
{
    sym *sp;
    if (sp = symtable.lookup($<s>2, 0)) {
        error_occurred = process_decl_err = true;
        sprintf(msgbuf,"redeclaration of process: %s", sp->name);
        errmsg.error(msgbuf);
    } else if (($<d>4 < 0 ) || ($<d>6 < 0) || ($<d>4 < $<d>6)) {
        error_occurred = process_decl_err = true;
        sprintf(msgbuf,"time error in process %s ", sp->name);
        errmsg.error(msgbuf);
    } else
        stack.push(symtable.add_process(stack.top_of(),
            $<s>2, $<d>4, 0.0));
}

} [TK_PROCESS process_name error ') ']
{
    error_occurred = process_decl_err = true;
    sprintf(msgbuf,"time error in process %s", $<s>2);
    errmsg.error(msgbuf);
}

;

process_name
:TK_IDENTIFIER
{

};

period
:time
{

};

time
:time_value
{

};
<table>
<thead>
<tr>
<th>time_value time_unit</th>
</tr>
</thead>
</table>
|$<d>$ = $<d>1 * $<d>2;
|
|;

**time_value**

: TK_DOUBLE

|$<d>$ = $<d>1;
|
|TK_INT

|$<d>$ = (double) $<i>1;
|
|;

**variance**

: time

|$<d>$ = $<d>1;
|
|;

**process_body**

:'{ process_statements '}'

|' [' ' ]'|

|;

**process_statements**

: process_statement

|process_statements process_statement |

|;

**process_statement**

: task_declaration

|family_definition |

|time_constraint |

|;

**task_declaration**

: TK_TASK task_declarator_list ';

|opt_specifier opt_task_declarator_list ';

|TK_TASK error ';

|error_occurred = true;

|sprintf(msgbuf,"task declaration error");

|errmsg.error(msgbuf);

|}

|opt_specifier error ';

|error_occurred = true;

|sprintf(msgbuf,"task declaration error");

|errmsg.error(msgbuf);

|}
task_declarator_list
  : task_declarator
  | task_declarator_list ',' task_declarator
  :

task_declarator
  : task_name '(' release_time ',' deadline ')' 
  |
  if (!process_decl_err) && (!family_decl_err) { 
    data_type *dp;
    char *name = words.slookup($<s>1);
    for (sym *sp = stack.top_of()->to_t;
         (sp != NILSYM) && (sp->name != name); sp = sp->next)
      :
    if (sp) 
      error_occurred = true;
      sprintf(msgbuf, "redeclaration of task: %s", name);
      errmsg.error(msgbuf);
    } 
  else 
    if ((dp = getdata.find_func(name)) == NULL) 
      error_occurred = true;
      sprintf(msgbuf, "task %s is not found as function", name);
      errmsg.error(msgbuf);
    } else if (dp->computation_time < 0) 
      error_occurred = true;
      sprintf(msgbuf, "computation time of task %s is unbounded", name);
      errmsg.error(msgbuf);
    } 
  else if (($<d>3 < 0) || ($<d>5 < 0) || ($<d>3 > $<d>5) 
    (dp->computation_time > ($<d>5 - $<d>3))) 
      error_occurred = true;
      sprintf(msgbuf, "time error of task %s", name);
      errmsg.error(msgbuf);
    } 
  else
    symtable.add_task(stack.top_of()->to_t, $<s>1, task, dp->filename,
                      $<d>3, $<d>5, dp->computation_time, 0, 0);
  }
  }
  :

task_name
  : TK_IDENTIFIER 
  |
  $<s>$ = $<s>1;
  }
  :

release_time
;time
{
  <$d>$ = <$d>1;
}
;

deadline
;time
{
  <$d>$ = <$d>1;
}
;

optSpecifier
:TK_SOFT_TASK
{
  task_option = soft_task;
}
|TK_CONT_TASK
{
  task_option = cont_task;
}
;

optTaskDeclaratorList
:optTaskDeclarator
|optTaskDeclaratorList ',' optTaskDeclarator
;

optTaskDeclarator
:taskName '(' release_time ',' deadline ',' allotment ',' variance ')' 
{
  if ((!process_decl_err) && (!family_decl_err)) {
    data_type *dp;
    char *name = words.slookup($<s>1);
    for (sym *sp = stack.top_of()->to_t;
        (sp != NILSYM) && (sp->name != name); sp = sp->next)
      ;
    if (sp) {
      errorOccurred = true;
      sprintf(msgbuf,"redeclaration of task: %s", name);
      errmsg.error(msgbuf);
    } else if (((dp = getData.find_func(name)) != NULL) {
      if (dp->computation_time >= 0){
        errorOccurred = true;
        sprintf(msgbuf,"computation time of task %s is bounded", name);
        errmsg.error(msgbuf);
      } else if ($<d>3 < 0) || ($<d>5 <0) || ($<d>3 > $<d>5) ||
        ($<d>7 < 0) || ($<d>9 <0)) {
        errorOccurred = true;
        sprintf(msgbuf,"time error of task %s", name);
        errmsg.error(msgbuf);
      } else if ($<d>3 < 0) || ($<d>5 <0) || ($<d>3 > $<d>5) ||
        ($<d>7 < 0) || ($<d>9 <0)) {
        errorOccurred = true;
        sprintf(msgbuf,"time error of task %s", name);
        errmsg.error(msgbuf);
      }
  } else {
    errorOccurred = true;
    sprintf(msgbuf,"process error of task %s", name);
    errmsg.error(msgbuf);
  }
};
else
    symtable.add_task(stack.top_of(), $<s>1, task_option,
    dp->filename, $<d>3, $<d>5, 0, $<d>7, $<d>9);

}
}
;

allotment
:time
{
    $<d>$ = $<d>1;
}
;

family_definition
:family_head family_body
{
    if ((!process_decl_err) && (!family_decl_err))
        stack.pop();
    else if (family_decl_err == family_nest)
        family_decl_err = false;

    family_nest—;
}
;

family_head
:TK_FAMILY family_name '(' occurrences ',' slack ')'
{
    family_nest++;
    if ((!process_decl_err) && (!family_decl_err)) {
        sym *sp;
        if (sp = symtable.lookup($<s>2, 1)) {
            family_decl_err = family_nest;
            error_occurred = true;
            sprintf(msgbuf,"redeclaration of family: %s", sp->name);
            errmsg.error(msgbuf);
        }
        else {
            register sym *p;
            double period;
            int occurrences = $<i>4;
            /* calculate the actual period of this family */
            for (p = stack.top_of(); p->cat != process; p = p->from)
                occurrences = occurrences * p->occurrences;
            period = p->period / occurrences;

            if (period < $<d>6) {
                family_decl_err = family_nest;
                error_occurred = true;
                sprintf(msgbuf, "out of lower bound of family: %s", $<s>2);
                errmsg.error(msgbuf);
            }
        }
    }
else if ((period + $<d>6) > (p->period + p->variance)) {
    family_decl_err = family_nest;
    error_occurred = true;
    sprintf(msgbuf, "out of upper bound of family: %s", $<s>2);
    errmsg.error(msgbuf);
} else
    stack.push(symtable.add_family(stack.top_of(),
        $<s>2, $<i>4, $<d>6));


} /*TK_FAMILY family_name '(', occurrences ')'*/

family_nest++;
if ( (!process_decl_err) && (!family_decl_err) ) {
    sym *sp;
    if (sp = symtable.lookup($<s>2, 1)) {
        family_decl_err = family_nest;
        error_occurred = true;
        sprintf(msgbuf,"redeclaration of family: %s", sp->name);
        errmsg.error(msgbuf);
    } else
        stack.push(symtable.add_family(stack.top_of(),
            $<s>2, $<i>4, 0.0));
}

/*TK_FAMILY family_name error ')'*/

family_nest++;
family_decl_err = family_nest;
error_occurred = true;
sprintf(msgbuf,"time error in family %s", $<s>2);
errmsg.error(msgbuf);

family_name
:TK_IDENTIFIER
    {$<s>$ = $<s>1;}
;

occurrences
:TK_INT
    {$<i>$ = $<i>1;}
;

slack
:time
    {
family_body
   ::= { family_statements }'
   | [ ]'

family_statements
   ::= family_statement
   | family_statements family_statement

family_statement
   ::= task_declaration
   | family_definition
   | time_constraint

time_constraint
   ::= task_identifier TK_PRECEDES task_identifier ';'
   { if (!process_decl_err) && (!family_decl_err) && (!task_rel_err) 
     symtable.add_relation(stack.top_of(), $<s>1, $<s>3, precedes); 
     task_rel_err = false; } 
   | task_identifier TK_EXCLUDES task_identifier ';'
   { if (!process_decl_err) && (!family_decl_err) && (!task_rel_err) 
     symtable.add_relation(stack.top_of(), $<s>1, $<s>3, excludes); 
     task_rel_err = false; } 
   | error ';'
   { error_occurred = true; 
     sprintf(msgbuf,"time relation error"); 
     errmsg.error(msgbuf); 
   }

task_identifier
   ::= task_name
   { if (!process_decl_err) && (!family_decl_err) && (!task_rel_err)) 
     char *name = words.slookup($<s>1); 
     for (sym *sp = stack.top_of()->to_t; 
       (sp != NILSYM) && (sp->name != name); sp = sp->next) 
       ; 
     if (sp == NILSYM) 
       error_occurred = task_rel_err = true; 
     if (stack.top_of()->cat == process) 
       sprintf(msgbuf, "task %s not defined in process %s", 

B - 10
name, stack.top_of()->name);
errmsg.error(msgbuf);
}
else {
  sprintf(msgbuf, "task %s not defined in family %s", 
    name, stack.top_of()->name);
  errmsg.error(msgbuf);
  return;
}
else
  $<s>$ = $<s>1;
}
}
}
else
  delete $<t>1;
}
}
}
}
}

families_identifier ':families_identifier_tail
{
  if (((!process_decl_err) && (!family_decl_err) &&(!task_rel_err))
    sym *sp;
    for (sp = $<t>1->p->to_t;

  if (sp == NILSYM) {
    error_occurred = task_rel_err = true;
    sprintf(msgbuf, "task %s not defined in family %s", 
      $<s>3, $<t>1->p->name);
    errmsg.error(msgbuf);
  }
  else {
    char buf[BufSize];
    sprintf(buf, "%s", $<t>1->s, $<s>3);
    $<s>$ = buf;
    delete $<t>1;
  }
}
}
}

families_identifier
  :families_identifier_tail
{
  if (((!process_decl_err) && (!family_decl_err) &&(!task_rel_err))
    $<t>$ = $<t>1;
}
}

families_identifier ':families_identifier_tail
{
  if (((!process_decl_err) && (!family_decl_err) &&(!task_rel_err))
    if ($<t>3->p->from != $<t>1->p) {
      error_occurred = task_rel_err = true;
      sprintf(msgbuf, "family %s not defined in family %s", 
        $<t>3->p->name, $<t>1->p->name);
      errmsg.error(msgbuf);
    }
    else {
      $<t>$ = new time_rel;
      $<t>$->p = $<t>3->p;
    }
  }
}
/* concatenate family names for task variable */
sprintf(($<t>$->s, "%s%s", $<t>1->s, $<t>3->s);
delete $<t>1;
delete $<t>3;
}
}
}
;

families_identifier_tail
:family_name ['index ']
|
|    if (!process_decl_err) && (!family_decl_err) && (!task_rel_err) { 
|        sym *sp;
|        if (sp = symtable.lookup($<s>1, 1)) { 
|            if (($<i>3 < 0) || ($<i>3 > sp->occurrences)) { 
|                error_occurred = task_rel_err = true;
|                sprintf(msgbuf, "out of index");
|                errmsg.error(msgbuf);
|            }
|            else { 
|                $<t>$ = new time_rel;
|                $<t>$->p = sp;
|                sprintf($<t>$->s, "%s%d", sp->name, $<i>3);
|            }
|        } 
|        else { 
|            error_occurred = task_rel_err = true;
|            sprintf(msgbuf, "family %s is not defined", $<s>1);
|            errmsg.error(msgbuf);
|        }
|    }
|}

index
:TK_INT
    |
    |
    |    $<i>$ = $<i>1;

%%
Lexical Analyzer for Script Translator

#include <stream.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "y.tab.h"
#include "error.hh"
#include "word.hh"
#include "sym.hh"
#include "getdata.hh"

#define yyerror(s) errmsg.error(s)
#undef yywrap
int yywrap();

ErrorMessage errmsg;
WordManager words;
extern SymTable symtable;
extern Stack stack;
extern GetData getdata;
ostream *fout;
filebuf f_out;

extern boolean error_occurred;
extern double default_time_unit;

static char msgbuf[BufSize];
static char out_file[BufSize];
static int Argc;
static char **argv;

static int id();
static int filename();
static int double_num();
static int int_num();
static void bad_char();
void comment_ppr();
void comment();
%

letter [a-zA-Z_]
digit [0-9]
alnum [a-zA-Z_0-9]

[ \s]* {errmsg.cnt_pos();} // ignore white spaces
\* {comment();}
\# {comment_ppr();}
main(int argc, char *argv[]) {
  int yyparse;
  void *f_open(char *);

  if (argc == 1)
    return 0;
  else if (argc == 2) {
    f_open(*++argv);
    return(yyparse());
  }
  else {
    Argc = argc - 1;
    Argv = argv + 1;
    f_open(*Argv);
    cerr << *Argv << ":\n";
    return(yyparse());
  }
}
if there is another input file, open it and continue,
otherwise stop parsing

```c
int yywrap()
{
    void f_open(char *);
    extern SymTable symtable;

    if (--Argc > 0) {
        (*fout).close(); // close the previous file
        delete fout;

        // reset global variables
        default_time_unit = 0.001;
        error_occurred = false;
        errmsg.reset();
        words.reset();
        symtable.reset();
        stack.reset();
        getdata.reset();

        f_open(*++Argv);
        cerr << *Argv << "\n";

        return 0;
    }

    return 1;
}
```

// open input file and output file, and reset global variables
void f_open(char *fp)
{
    FILE *Fp;
    extern FILE *yyin;
    char *tmp;

    if ((Fp = fopen(fp, "r")) == NULL) {
        cerr << "file " << fp << " cannot be opened\n";
        exit(1);
    }
    fclose(Fp);

    // check legal source file name .rts
    for (tmp = fp; *++tmp != 0; )
    {
        if (strcmp(tmp - 4, ".rts")) {
            cerr << "unknown type of file " << fp << " (bad magic number)\n";
            exit(1);
        }
    }
    sprintf(msgbuf, "cc -E %s", fp);
    yyin = popen(msgbuf, "r");

    // generate output file name .score
tmp = (tmp = strrchr(fp, '/'))? tmp+1: fp;
sprintf(out_file, "%s", tmp);
out_file[strlen(tmp)-3] = '0';
strcat(out_file, "score");

if (!f_out.open(out_file, output)) {
    cerr << "file " << out_file << " cannot be opened\n";
    exit(1);
}
fout = new ostream(&f_out); // open output file(.score)
}

static int id()
{
    yylval.s = words.lookup(yytext);
    return TK_IDENTIFIER;
}

// get the source file name ("fn")
static int filename()
{
    yytext[yyleng-1] = '0';

    // checking legal source file name
    if ((yytext[yyleng-3] != '.') || (yytext[yyleng-2] != 'c'))
    {
        error_occurred = true;
        sprintf(msgbuf, "unknown type of file %s", yytext+1);
        errmsg.error(msgbuf);
        yylval.s = NULL;
    }
    else
    { yylval.s = yytext + 1;
        return TK_FILENAME;
    }
}

static int double_num()
{
    sscanf(yytext, "%lf", &yylval.d);
    return TK_DOUBLE;
}

static int int_num()
{
    sscanf(yytext, "%d", &yylval.i);
    return TK_INT;
}

// read remainder of comments
void comment()
{
    char c;
    c = yyinput();
do {

unput(c);
while ((c = yyinput()) != '*')
    if (c) {
        error_occurred = true;
        errmsg.error("unterminated comment");
    } else if (c == '\n') {
        errmsg.linenumber++;
        errmsg.columnnumber = 0;
    } else if (c == '\t')
        errmsg.columnnumber += 8 - (errmsg.columnnumber % 8);
    else
        errmsg.columnnumber++;
    if (c)
        errmsg.columnnumber++; // count the '*'
} while ((c = yyinput()) && (c != '/'));
if (!c) {
    error_occurred = true;
    errmsg.error("unterminated comment");
} else
    errmsg.columnnumber++; // count the closing '/'

// read remainder of comments made by preprocessor
void comment_ppr()
{
    int line = 0, i = 0;
    char c;
    yyinput(); // pop up " "
    while (isdigit(c = yyinput()))
        line = 10 * line + (c - '0');
    errmsg.linenumber = line - 1;
    if (c != '\n') {
        errmsg.filename[i++] = yyinput();
        while (errmsg.filename[i++] = yyinput()) != '\n')
            ;
        errmsg.filename[i] = '\0';
    }
}

// unrecognized character found
static void bad_char()
{
    sprintf(msgbuf, "unknown character 0%o (%c)", yytext[0], yytext[0]);
    error_occurred = true;
    errmsg.error(msgbuf);
}
(word.hh)

#define MaxWords 509
#define MaxBuf 4096

// permanent word management routine

class WordManager {
private:
  char *wptr[MaxWords];       // a fixed array of word pointers
  char buf[MaxBuf];
  char *nxtfree;
  unsigned hash(char *);

public:
  WordManager();
  void reset();
  char *slookup(char *);
};
(word.cc)

#include <string.h>
#include "word.hh"

// treat wptr as a hash table
// all entries are initially NULL
// entries that are in use are not NULL
// on collision, search forward to an empty entry in the array

WordManager::WordManager()
{
    nxtfree = buf;

    for (register i = 0; i < MaxWords; i++)
        wptr[i] = NULL;
}

void WordManager::reset()
{
    nxtfree = buf;

    for (register i = 0; i < MaxWords; i++)
        wptr[i] = NULL;
}

// hash function
unsigned WordManager::hash(char *s)
{
    int i = 0;

    while (*s)
        i = i * 16 + *s++;

    return i;
}

// look up the specified string, returning a pointer to it
// or null if not found
char *WordManager::slookup(char *s)
{
    int h;

    h = hash(s) % MaxWords;
    while (wptr[h] && strcmp(s, wptr[h]))
        h = (h + 1) % MaxWords;
    if (!wptr[h]) {
        wptr[h] = nxtfree;

        // still need to check for overflow
        while (*nxtfree++ = *s++)
            ;
    }

    return wptr[h];
}
```c
#define HASH_SIZE 17
#define STACK_SIZE 512
#define BufSize 512
#define NILSYM ((struct sym *) 0)
#define NILREL ((struct rel_list *) 0)
#define NILNAM ((struct rel_name_list *) 0)

enum categories { process, family, task, soft_task, cont_task };
enum rel_cat { precedes, excludes };
enum boolean { false, true };

// family name list for task name
struct rel_list {
    char rel[BufSize];
    rel_list *next;
};

// symbol table structure:
// process node:family node: task node:
// name name name
// cat cat cat
// period occurrences filename
// variance slack release_time
// to_f to_f deadline
// to_t to_t computation_time
// next from allotment
// nxtsym next variance
// nxtsym from
// next
// nxtsym
struct sym {
    char *name;
    categories cat;
    char *filename;
    double period;
    int occurrences;
    double variance;
    double slack;
    double release_time;
    double deadline;
    double computation_time;
    double allotment;
    sym *to_f;
    sym *to_t;
    sym *from;
    sym *next;
    sym *nxtsym;
    rel_list *rellist;
};

// timing relation list in a process or family
struct rel_name_list {
    sym *sp;
    rel_name_list *next;
}
```
// symbol table
class SymTable {
private:
    sym *stable[HASH_SIZE]; // symbol table uses hashing
    sym *free_sym;            // free list of symbol node
    rel_list *free_rel_list;
    rel_name_list *free_rel_name_list;
    void pr_tasks(sym *, char *);
    void pr_variables(sym *, char *);
    void pr_relations(sym *);
    void add_relation_(rel_list *, rel_name_list *,
                       char *, char *, rel_cat, char *);

public:
    SymTable();
    void reset();
    sym *lookup(char *name, int);
    sym *add_process(sym *, char *, double, double);
    sym *add_family(sym *, char *, int, double);
    void add_task(sym *, char *, categories, char *,
                   double, double, double, double, double);
    void add_relation(sym *, char *, char *, rel_cat);
    void pr_process(sym *);
    void pr_variables(sym *);
    void pr_relations_rec(sym *, char *, int, boolean);
    void remove_all();
};

// stack of current pointer of process or family node
class Stack {
private:
    sym *s[STACK_SIZE];
    int top;
public:
    Stack();
    void reset();
    void push(sym *
    sym *pop();
    sym *top_of();
};
SymTable::SymTable()
{
    register int i;
    for (i = 0; i < HASH_SIZE; i++)
    {
        stable[i] = NILSYM;
        free_sym = NILSYM;
        free_rel_list = NILREL;
        free_rel_name_list = NILNAM;
    }

    // initialize symbol table
    void SymTable::reset()
    {
        register int i;
        for (i = 0; i < HASH_SIZE; i++)
            if (stable[i] != NILSYM) {
                stable[i]->nxtsym = free_sym;
                free_sym = stable[i];
                stable[i] = NILSYM;
            }
    }

    // look up the specified variable name, returning a pointer to the
    // symbol table or null if not found
    sym *SymTable::lookup(char *name, int flag)
    {
        if (name = words.slookup(name))
        {
            register int hash;
            register sym *sp;

            hash = ((int) name) % HASH_SIZE;

            // if flag is 0, checking for process,
            // otherwise for family.
            if (!flag) {
                for (sp = stable[hash]; sp; sp = sp->nxtsym)
                    if (sp->name == name)
                        return sp;
        }
else {
    for (sp = stable[hash]; sp; sp = sp->nxtsym)
        if ((sp->name == name) && (sp->cat == family))
            return sp;
    }
return NULL;
}

// create a new symbol structure of process node
sym *SymTable::add_process(sym *pptr, char *name,
                            double period, double variance)
{
    register int hash;
    register sym *sp;

    if ((sp = free_sym) == NULL) {
        if ((sp = (sym *) malloc(sizeof (sym))) == NULL) {
            sprintf(msgbuf, "out of malloc space");
            errmsg.fatal(msgbuf);
        }
    }
    else
        free_sym = sp->nxtsym;

    sp->name = words.slookup(name);
    sp->cat = process;
    sp->period = period;
    sp->variance = variance;
    sp->to_f = NILSYM;
    sp->to_t = NILSYM;
    sp->next = NILSYM;
    sp->rellist = NILREL;

    hash = ((int) sp->name) % HASH_SIZE;
    sp->nxtsym = stable[hash];
    stable[hash] = sp;

    if (pptr != NILSYM)
        pptr->next = sp;

    return sp;
}

// create a new symbol structure of family node
sym *SymTable::add_family(sym *fptr, char *name,
                          int occurrences, double slack)
{
    register int hash;
    register sym *sp;

    if ((sp = free_sym) == NULL) {
        if ((sp = (sym *) malloc(sizeof (sym))) == NULL) {
            sprintf(msgbuf, "out of malloc space");
            errmsg.fatal(msgbuf);
        }
    }

else
    free_sym = sp->nxtsym;

sp->name = words.slookup(name);
sp->cat = family;
sp->occurrences = occurrences;
sp->slack = slack;
sp->to_f = NILSYM;
sp->to_t = NILSYM;
sp->from = fptr;
sp->next = NILSYM;
sp->rellist = NILREL;

hash = ((int) sp->name) % HASH_SIZE;
sp->nxtsym = stable[hash];
stable[hash] = sp;

if (fptr->to_f == NILSYM)
    fptr->to_f = sp;
else {
    sym *ptr = fptr->to_f;
    while (ptr != NILSYM) {
        fptr = ptr;
        ptr = ptr->next;
    }
    fptr->next = sp;
}

return sp;

// create a new symbol structure of task node
void SymTable::add_task(sym *fptr, char *name, categories cat,
                         char *filename, double release_time,
                         double deadline, double computation_time,
                         double allotment, double variance)
{

    register int hash, i;
    register sym *sp;
    char resolved_path[MAXPATHLEN], name_buf[BufSize], *name_tmp;

    if ((sp = free_sym) == NULL) {
        if ((sp = (sym *) malloc(sizeof (sym))) == NULL) {
            sprintf(msgbuf, "out of malloc space");
            errmsg.fatal(msgbuf);
        }
    }
    else
        free_sym = sp->nxtsym;

    sp->name = words.slookup(name);
    sp->cat = cat;
strcpy(name_buf, filename);
*(name_buf + strlen(name_buf) - 1) = 'o';  // change suffix .i into .o
realpath(name_buf, resolved_path);
name_tmp = new char[BufSize];
strcpy(name_tmp, resolved_path);
sp->filename = name_tmp;
sp->release_time = release_time;
sp->deadline = deadline;
sp->computation_time = computation_time;
sp->allotment = allotment;
sp->variance = variance;
sp->from = fptr;
sp->next = NILSYM;

hash = ((int) sp->name) % HASH_SIZE;
sp->nxtsym = stable[hash];
stable[hash] = sp;

if (fptr->to_t == NILSYM)
    fptr->to_t = sp;
else {
    sym *ptr = fptr->to_t;
    while (ptr != NILSYM) {
        fptr = ptr;
        ptr = ptr->next;
    }
    fptr->next = sp;
}

// remove the symbol table entry for the family node and the task node
// and free them
void SymTable::remove_all()
{
    register int i;

    for (i = 0; i < HASH_SIZE; i++) {
        tmplist = NILSYM;
        for (sp = stable[i]; sp != NILSYM; sp = nextsp) {
            nextsp = sp->nxtsym;
            if (sp->cat == process) {
                sp->to_f = NILSYM;
                sp->to_t = NILSYM;
                sp->nxtsym = NILSYM;

                // insert sp into temporary list
                if (tmplist == NILSYM)
                    tmplist = sp;
                else {
                    for (tmp = tmplist; tmp; tmp = tmp->nxtsym)
                        prevtmp = tmp;
                    prevtmp->nxtsym = sp;
                }
            }
        }
    }
}
else {
    if (sp->cat != family) // if task or opt_task, free memory allocated
        delete sp->filename; // for its full path name
    sp->nxtsym = free_sym;
    free_sym = sp;
}
}

stable[i] = tmplist;
}
}

// recursive function of add_relation
void SymTable::add_relation(rel_list *lptr, rel_name_list *nptr,
                           char *tname1, char *tname2,
                           rel_cat rcat, char *family_name)
{
    register rel_list *lp, *llp;
    char buf[BufSize];

    if (nptr == NILNAM) {
        if ((lp = free_rel_list) == NULL) {
            if ((lp = (rel_list *) malloc(sizeof(rel_list))) == NULL) {
                sprintf(msgbuf, "out of malloc space");
                errmsg.fatal(msgbuf);
            }
        }
    } else
        free_rel_list = lp->next;

    if (rcat == precedes)
        sprintf(lp->rel,"(%s%s;%s%s)", family_name, tname1, family_name, tname2);
    else // excludes
        sprintf(lp->rel,"(%s%s!%s%s)", family_name, tname1, family_name, tname2);

    lp->next = NILREL;
    for (llp = lptr; llp->next != NILREL; llp = llp->next)
    {
        llp->next = lp;
    }
    else {
        for (register int i = 0; i < nptr->sp->occurrences; i++) {
            sprintf(buf,"%s%s$%d", family_name, nptr->sp->name, i);
            add_relation_(lptr, nptr->next, tname1, tname2, rcat, buf);
        }
    }
}

// create timing relation structure and add it to the relation list
void SymTable::add_relation(sym *sptr, char *tname1, char *tname2,
                           rel_cat rcat)
{
    register sym *sp;
    register rel_list *lp;
    register rel_name_list *rp, *nptr;
}
nptr = NILNAM;
for (sp = sptr; sp->cat != process; sp = sp->from) {
    if ((rp = free_rel_name_list) == NULL) {
        if ((rp = (rel_name_list *) malloc(sizeof (rel_name_list))) == NULL) {
            sprintf(msgbuf, "out of malloc space");
            errmsg.fatal(msgbuf);
        }
    } else
        free_rel_name_list = rp->next;
    rp->sp = sp;
    rp->next = nptr;
    nptr = rp;
}
if ((lp = free_rel_list) == NULL) {
    if ((lp = (rel_list *) malloc(sizeof (rel_list))) == NULL) {
        sprintf(msgbuf, "out of malloc space");
        errmsg.fatal(msgbuf);
    }
} else
    free_rel_list = lp->next;
lp->next = NILREL;
add_relation_(lp, nptr, tname1, tname2, rcat, "");

// free nptr
if (free_rel_name_list == NILNAM)
    free_rel_name_list = nptr;
else {
    for (rp = free_rel_name_list; rp->next != NILNAM; rp = rp->next)
        ;
    rp->next = nptr;
}
if (sptr->rellist == NILREL)
    sptr->rellist = lp->next;
else {
    for (register rel_list *llp = sptr->rellist;
         llp->next != NILREL; llp = llp->next)
        ;
    llp->next = lp->next;
}

// release lp
lp->next = free_rel_list;
free_rel_list = lp;

// print task name
void SymTable::pr_tasks(sym *fptr, char *family_name)
{
    register sym *sp;
    char buf[BufSize];

    if (fptr != NILSYM) {
        }
for (sp = fptr->to_t; sp != NILSYM; sp = sp->next)
if (sp->from->cat == process) {
    if (sp->cat == task)
        *fout << " " << family_name << sp->name << "(" << sp->filename << "," << sp->computation_time << "," << sp->release_time << ";" << sp->deadline << ")\n";
else if (sp->cat == soft_task)
        *fout << " soft " << family_name << sp->name << "(" << sp->filename << "," << sp->allotment << ";" << sp->variance << "," << sp->release_time << ";" << sp->deadline << ")\n";
else // cont_task
        *fout << " cont " << family_name << sp->name << "(" << sp->filename << "," << sp->allotment << ";" << sp->variance << "," << sp->release_time << ";" << sp->deadline << ")\n";
    } else {
if (sp->cat == task)
        *fout << " " << family_name << sp->name << "(" << sp->filename << "," << sp->computation_time << "," << family_name << "+" << sp->release_time << "+" << family_name << "+" << sp->deadline << ")\n";
else if (sp->cat == soft_task)
        *fout << " soft " << family_name << sp->name << "(" << sp->filename << "," << sp->allotment << ";" << sp->variance << "," << family_name << "+" << sp->release_time << "+" << family_name << "+" << sp->deadline << ")\n";
else // cont_task
        *fout << " cont " << family_name << sp->name << "(" << sp->filename << "," << sp->allotment << ";" << sp->variance << "," << family_name << "+" << sp->release_time << "+" << sp->deadline << ")\n";
    }
}

for (sp = fptr->to_f; sp != NILSYM; sp = sp->next) {
for (int i = 0; i < sp->occurrences; i++) {
    sprintf(buf,"%s%$%d", family_name, sp->name, i);
    pr_tasks(sp, buf);
}
}

// recursive function of pr_variable
void SymTable::pr_variables_rec(sym *fptr, char *family_name, int num_occr, boolean last_occr)
{
    register sym *sp;
    register int i;
    register boolean last_occrr_flag;
    char buf[BufSize];

    while (fptr != NILSYM) {
        ...
for (i = 0; i < fptr->occurrences; i++) {
    if (fptr->from->cat == process) {
        if (i == 0) {
            #fout << " " << fptr->name << "$" << 0 << ":" << 0
            << ")\n";
        }
        else {
            #fout << " " << fptr->name << "$" << i << "("
            << fptr->name << "$" << i-1 << ":4"
            << fptr->from->name << "$/" << fptr->occurrences << ":-"
            << fptr->slack << ":";
            #fout << " " << fptr->name << "$" << i-1 << "$/" << fptr->from->name << ":-"
            << fptr->slack << ":";
            if (i == (fptr->occurrences-1)) {
                #fout << " " << fptr->name << "$" << i-1 << "($" << fptr->name << "$/" << fptr->occurrences << ":-"
                << fptr->slack << ":";
            }
            else {
                #fout << " " << fptr->name << "$" << i << "("
                << fptr->from->name << "$/" << fptr->occurrences << ":-"
                << fptr->slack << ":";
                if (i == (fptr->occurrences-1)) {
                    #fout << " " << fptr->name << "$" << i-1 << "($" << fptr->name << "$/" << fptr->occurrences << ":-"
                    << fptr->slack << ":";
                }
                else {
                    if (i == (fptr->occurrences-1)) {
                        last_occr_flag = true;
                    } else {
                        last_occr_flag = false;
                    }
                    sprintf(buf, ":$", fptr->name);
                    pr_variables_rec(fptr->to_f, buf, i, last_occr_flag);
                }
            }
        }
    } else {
        if (i == 0) {
            #fout << " " << family_name << num_occr
            << fptr->name << "$" << i << "("
            << family_name << num_occr << ":-
            << family_name << num_occr << ");\n";
        }
        else {
            #fout << " " << family_name << num_occr
            << fptr->name << "$" << i-1 << "("
            << sp->name << "$" << num_occr << ":-
            << sp->name << "$" << num_occr << ");\n";
            if (i == (fptr->occurrences-1)) {
                *fout << " " << family_name << num_occr
                << fptr->name << "$" << i-1 << "($" << sp->name << "$" << num_occr
                << fptr->slack << ":-
                << fptr->slack << ");\n";
            }
            else {
                for (sp = fptr->from ;sp->cat != process; sp = sp->from);
                #fout << " " << family_name << num_occr
                << fptr->name << "$" << i << "("
                << family_name << num_occr << fptr->name << "$" << i-1 << "+("
                << sp->name << "$" << num_occr << family_name << num_occr
                << ")\n";
                #fout << " " << family_name << num_occr
                << fptr->name << "$" << i-1 << "$/" << sp->name << ":-"
                << family_name << num_occr << ");\n";
                if (i == (fptr->occurrences-1)) {
                    *fout << " " << family_name << num_occr
                    << fptr->name << "$" << i << "($" << sp->name << "$" << "+("
                    << sp->name << "$" << num_occr << family_name << num_occr
                    << ")\n";
                }
            }
        }
    }
}
if (last_occr && (i == (fptr->occurrences-1)))
    last_occr_flag = true;
else
    last_occr_flag = false;
sprintf(buf, ":$", fptr->name);
pr_variables_rec(fptr->to_f, buf, i, last_occr_flag);
}
last_occ_flag = true;
}
else {
*fout << " " << family_name << num_occrr
<< fptr->name << "$" << i << "("
<< family_name << num_occrr << fptr->name << "$" << i-1
<< "+" << family_name << num_occrr+1 << "="
<< family_name << num_occrr
<< ")" << fptr->occurrences << "="
<< fptr->slack << "."
<< family_name << num_occrr << fptr->name << "$" << i-1
<< "+" << family_name << num_occrr+1 << "="
<< family_name << num_occrr
<< ")" << fptr->occurrences << "+
<< fptr->slack << ")";
if (i == (fptr->occurrences-1)) {
*fout << " " << family_name << num_occrr
<< fptr->name << "$" << i << "(
<< family_name << num_occrr+1 << "="
<< family_name << num_occrr+1 << "="
<< family_name << num_occrr
<< ")" << fptr->occurrences << "="
<< fptr->slack << "."
<< family_name << num_occrr+1 << "="
<< family_name << num_occrr+1 << "="
<< family_name << num_occrr
<< ")" << fptr->occurrences << "+
<< fptr->slack << ")";
}
last_occ_flag = false;
}
}
sprintf(buf, "%s%d%s$", family_name, num_occrr, fptr->name);
pr_variables_rec(fptr->to_f, buf, i, last_occ_flag);
}
}
fptr = fptr->next;
}
}

// print variables used in listing tasks
void SymTable::pr_variables(sym *pptr)
{
if (pptr != NILSYM) {
*fout << " " << pptr->name << "$" << "("
<< pptr->period - pptr->variance << ":"
<< pptr->period + pptr->variance << ")";
pr_variables_rec(pptr->to_f, ",", 0, true );
}
// print timing relation of tasks
void SymTable::pr_relations(sym *fptr)
{
    if (fptr != NILSYM) {
        register sym *sp;
        register rel_list *rp;

        for (rp = fptr->rellist; rp != NILREL; rp = rp->next)
            *fout << " " << rp->rel << "\n";

        // release fptr->rellist
        if (free_rel_list == NILREL) {
            free_rel_list = fptr->rellist;
            fptr->rellist = NILREL;
        } else {
            for (rp = free_rel_list; rp->next != NILREL; rp = rp->next)
                rp->next = fptr->rellist;
            fptr->rellist = NILREL;
        }

        for (sp = fptr->to_f; sp != NILSYM; sp = sp->next)
            pr_relations(sp);
    }
}

// print output for a process
void SymTable::pr_process(sym *pptr)
{
    if (pptr != NILSYM) {
        *fout << "process " << pptr->name << "\n";

        *fout << "# timing constraints\n";
        pr_tasks(pptr, "");

        *fout << "\n # variables\n";
        pr_variables(pptr);

        *fout << "\n# task relations\n";
        pr_relations(pptr);
    }
}

Stack::Stack()
{
    top = 0;
    s[top] = NILSYM;
}

void Stack::reset()
{
    top = 0;
    s[top] = NILSYM;
}
void Stack::push(sym *p)
{
    if (++top == STACK_SIZE) {
        sprintf(msgbuf, "out of stack size");
        errmsg.fatal(msgbuf);
    }
    else s[top] = p;
}

sym *Stack::pop()
{
    if (top < 0) {
        sprintf(msgbuf, "out of stack size");
        errmsg.fatal(msgbuf);
    }
    else return (s[top--]);
}

sym *Stack::top_of()
{
    return (s[top]);
}
(getdata.hh)

#define NILFILE ((struct file_list *) 0)
#define BufSize 512

struct file_list {
  char *file;
  file_list *next;
};

struct data_type {
  char *filename;
  double computation_time;
};

// communicator with the source, intermediate file of real time C-compiler
class GetData {
private:
  file_list *flist;
  file_list *free_flist;
public:
  GetData();
  void reset();
  void execute(char *);
  data_type *find_func(char *);
};
**(getdata.cc)**

```c
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include "getdata.hh"
#include "error.hh"

extern ErrorMessage errmsg;

static char msgbuf[BufSize];

data_type data;

GetData::GetData()
{
    flist = NILFILE;
    free_flist = NILFILE;
}

void GetData::reset()
{
    free_flist = flist;
    flist = NILFILE;
}

void GetData::execute(char *filename)
{
    struct stat stat_buf;
    register file_list *fp;
    register int i;

    // check time-stamp of source file
    if (stat(filename, &stat_buf) == 0) {
        if ((stat_buf.st_size > 0) && (stat_buf.st_atime <= stat_buf.st_mtime)) {
            sprintf(msgbuf, "rtcc -c %s", filename);
            system(msgbuf); // call real time C compiler
        }
    }

    if ((fp = free_flist) == NILFILE) {
        if ((fp = (file_list *) malloc(sizeof (file_list))) == NILFILE) {
            sprintf(msgbuf, "out of malloc space");
            errmsg.fatal(msgbuf);
        }
    }
    else
        free_flist = fp->next;

    fp->file = filename;
    *(fp->file + strlen(fp->file) - 1) = 'i'; // change suffix .c into .i

    fp->next = NILFILE;

    if (flist == NILFILE)
        flist = fp;
```
else {
    fp->next = flist;
    flist = fp;
}
}

data_type *GetData::find_func(char *task_name) {
    register file_list *fp;
    FILE *fin;
    char func_name[BufSize];

    for (fp = flist; fp != NILFILE; fp = fp->next) {
        if ((fin = fopen(fp->file, "r")) != NULL) {
            fscanf(fin, "%s", func_name);
            while (strcmp(func_name, "\$\$")) {
                if (!memcmp(func_name, task_name, strlen(task_name))) {
                    data.filename = fp->file;
                    if (!strcmp((strchr(func_name, '$') + 1), "bound")) {
                        getc(fin); getc(fin); // read out "=
                        fscanf(fin, "%lf", &(data.computation_time));
                        fclose(fin);
                        return &data;
                    } else {
                        data.computation_time = -1;
                        fclose(fin);
                        return &data;
                    }
                } else {
                    while(getc(fin) != 'n')
                        // discard the rest characters in the line
                    fscanf(fin, "%s", func_name);
                }
            }
        } else
            while(getc(fin) != '
'); // discard the rest characters in the line
        fscanf(fin, "%s", func_name);
    }
}

if (fp == NILFILE)
    return NULL;
#define MAX_ERROR 5
#define BufSize 512

// error message manager
class ErrorMessage {
private:
    char filename[BufSize];
    int linenumber, columnnumber;
    int errcount;
public:
    ErrorMessage();
    void reset();
    void cnt_pos();
    void error(char *);
    void warning(char *);
    void fatal(char *);
    friend void comment();
    friend void comment_ppr();
};
#include <stream.h>
#include "error.hh"

ErrorMessage::ErrorMessage()
{
    linenumber = 1;
    columnnumber = 0;
    errcount = 0;
}

void ErrorMessage::reset()
{
    linenumber = 1;
    columnnumber = 0;
    errcount = 0;
}

// keep column number and line number correct
void ErrorMessage::cnt_pos()
{
    int i;
    extern char *yytext;

    for (i = 0; yytext[i] != '\0'; i++)
        if (yytext[i] == '\n') {
            linenumber++;
            columnnumber = 0;
        } else if (yytext[i] == '\t')
            columnnumber += 8 - (columnnumber % 8);
        else
            columnnumber++;
}

void ErrorMessage::error(char *msg)
{
    errcount++;
    cerr << "rtsc: " << filename << ", error on line " << linenumber
         << ", col " << columnnumber << ": " << msg << "\n";

    if (errcount >= MAX_ERROR) {
        cerr << "rtsc: sorry, cannot recover from previous errors\n";
        exit(1);
    }
}

void ErrorMessage::warning(char *msg)
{
    cerr << "rtsc: " << filename << ", warning on line " << linenumber
         << ", col " << columnnumber << ": " << msg << "\n";
}

void ErrorMessage::fatal(char *msg)
{
cerr << "rtsc: " << filename << ", fatal error: " << msg << ";
exit(1);
# TABLE OF CONTENTS

1. Introduction ........................................................................................................... 1
2. System Overview .................................................................................................... 3
3. Survey of Related Research .................................................................................. 5
4. Description of rtScript ............................................................................................ 7
   4.1: Language Overview ......................................................................................... 7
   4.2: Structure of rtScript ....................................................................................... 8
      4.2.1: rtC Source File Declaration .................................................................... 9
      4.2.2: Time Units .............................................................................................. 10
      4.2.3: Process Statements ................................................................................. 10
      4.2.4: Family Statements .................................................................................. 11
      4.2.5: Task Statements ..................................................................................... 12
      4.2.6: Task Relationships .................................................................................. 14
   4.3: Using rtScript ................................................................................................... 15
5. Overview of Implementation ................................................................................... 16
   5.1: The Parser and Lexical Analyzer ................................................................... 16
   5.2: The Script Translator ..................................................................................... 16
6. Discussion of the Output of rtScript ....................................................................... 17
   6.1: Structure of the Output .................................................................................. 17
   6.2: Timing Constraints ........................................................................................ 18
   6.3: Timing Relationships ...................................................................................... 23
7. Further Research .................................................................................................... 26
8. Acknowledgement .................................................................................................. 27
Bibliography ............................................................................................................... 28
TABLE OF CONTENTS – Continued

Appendix A: Example rtScript Program ........................................ A-1
Appendix B: Source Code of the rtScript Translator ......................... B-1