Expressing and Maintaining Timing Constraints in FLEX

Kwei-Jay Lin
Swaminathan Natarajan

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

Abstract
Real-time programs should be able to specify their timing and resource constraints. We describe the timing constraint mechanism in a real-time programming language called FLEX. A FLEX program may use the constraint primitives to express timing and resource requirements. If the required time or resources are not available at run-time, a FLEX program may dynamically produce monotonic imprecise results. Both time and system resources are defined as first-class objects in the language so that they can be evaluated just like any other first-class object. By unifying time, resources and normal objects, the semantics and the executions of real-time programs are more manageable. We also discuss some implementation issues of FLEX and present some performance data.

1. Introduction
To ensure that a real-time program meets specifications, the language used to implement the program should allow different types of timing requirements to be easily stated, and also provide ways to guarantee that the requirements will always be met. Most classical real-time programming languages (e.g. Modula [25], Edison [11]) treat time as a system object which is not relevant to the program logic but only monitored by the scheduler. In such languages, a program cannot change its required execution time, hence being able to specify it is not very useful. More recent real-time language designs allow some timing constraints to be specified in order to trigger timing events. But most of them fall short of ensuring that constraints are always satisfied. Using the model of imprecise computation [15,18], it is possible for a program to dynamically modify its execution time. In other words, time can be treated as a first-class object in real-time programs since it can be manipulated just like a normal object. And since the execution time often depends on the system resources used (e.g. processor, communication, etc.), resources can be treated in the same way as well. We thus propose a constraint mechanism which unifies the semantics of time, resources and normal objects.

In this paper, we discuss the issues of expressing and maintaining timing constraints in the design of a language called FLEX [20]. Conventional approaches to implementing real-time systems often design programs to handle the worst-case situation. Consequently, the systems implemented are usually excessively configured for the average case. More importantly, they may not handle dynamic situations and abnormal exceptions as required in fault-tolerant systems. FLEX allows programmers to implement programs which are flexible in the time required for execution. Two mechanisms are provided: a constraint mechanism for expressing timing requirements and a control mechanism for maintaining the validity of constraints. In this paper, we describe how different types of constraints may be expressed in FLEX, how they may be satisfied in a flexible way using the concept of imprecision, and how imprecision in values and computations may be represented and processed.

The rest of the paper is organized as follows. Section 2 discusses different approaches that have been used to program real-time systems and the issues in providing a time constraint mechanism. The constraint mechanism of FLEX is presented in Section 3. Section 4 outlines how imprecise computations can be used to guarantee constraints. Implementation issues are discussed in section 5. We present our conclusions in section 6.

2. Timing constraints in real-time programs
Many approaches to programming real-time systems have been proposed. The first approach is to program in assembler or other low level languages. Exhaustive trial-and-error testing is employed to establish that programs meet the timing requirements [9]. A second approach is to use languages such as Ada [7] and Modula which are primarily general-purpose programming languages. The programmer uses mechanisms such as coroutines, process switching, priorities, interrupts and exception handling to modify the execution behavior. Not only is it awkward to write real-time programs in languages which do not permit explicit expression of timing requirements [1], but
also the programs implemented are sensitive to hardware characteristics and system configurations.

The third approach, adopted in recent language proposals like ESTEREL [2], DPS [14] and Real-Time Euclid [13], permits direct expression of timing constraints. The program specifies the deadlines for procedures. However, to make run-time enforcement possible, each language adopts some specific measures. ESTEREL defines the language to be synchronous, i.e. all events are instantaneous and executions of basic operations take no time. DPS provides an exception handling mechanism to cope with timing errors. Real-Time Euclid restricts the language constructs to those which are time-bounded. The programmer may not use recursion, dynamic memory allocation or dynamic process instantiation because their execution time is unpredictable. These restrictions make it easier to estimate the execution time of the program, and to provide schedules which meet deadlines.

Two separate issues are involved in a timing constraint mechanism: First, what primitives should be provided for expressing timing constraints, and do they form a complete set? Second, if the timing constraints cannot be met, how does the program respond to it? In this section, we review the two issues.

2.1. Expressing constraints

Much work has been done in specifying timing requirements. Although the requirement can be specified both at the program specification level [12] and at the program operation level, in this paper we are only interested in primitives at the operation level. In general, simplicity and completeness are two major goals in providing primitives for expressing timing constraints. In [5], two categories of timing constraints are identified: performance constraints which set limits on the response time, and behavioral constraints which make demands on the rates of stimuli. For each constraint, three types of restrictions are further identified: maximum, minimum and duration. As discussed above, several languages have included timing requirement primitives in their designs. Due to its synchronous nature, ESTEREL requires only one temporal scope construct for all categories and types of constraints: the up_to statement. In [2], the completeness of the statement is suggested since it may simulate all other timing constructs such as every, from .. to etc. DPS utilizes three temporal scope structures: internal, repetitive and consecutive. Each structure may specify three timing parameters: delay, execute and deadline.

To get an accurate timing analysis, many architectural, operating system and language factors must be specified or defined [10]. In order to achieve the best performance from a computation, factors in time and space are often closely related. When certain space (or resource) requirements cannot be satisfied, many algorithms allow a computation to spend more time to produce results of the same quality. On the other hand, when computing resources are abundant, a computation may be able to speed up its processing by utilizing more resources concurrently or advantageously. Thus it is often desirable or even necessary to specify the requirements for resources, in addition to time, in real-time programs. However, very little work has been done in language designs in expressing the constraints for architectural and operating system resources required for program execution.

Another reason why resource requirements should be specified in the program is that the resources available in many systems are not fixed. Modern real-time systems must be fault-tolerant since they may operate in isolation for a long time (e.g. submarines or space vehicles). Moreover, certain system components may be temporarily removed from service in order to conduct periodic maintenance or replacements. In such cases, the resources available for program execution will be less than normal. But the system must continue to provide a timely response to all important events and to maintain normal operation. Most real-time programs which have been implemented do not work with reduced resources. Many other systems treat resources as operating system parameters and ignore them in the program level. To make programs flexible and configuration-independent, a real-time program should allow resources, just like time, to be handled as first-class objects. FLEX unifies both with other normal objects in the system. We will present our design in Section 3.

2.2. Maintaining Constraints

Timing constraints express the requirements for time. But by themselves, they merely detect instances when the condition is not satisfied. There is still a need for language features to enforce timing constraints and to ensure program completion before the deadline. To illustrate our concept of maintaining timing constraints, we use a simple example of a moving robot as follows:

```plaintext
loop every 1 second {
    Interpret vision signals and
    check for hazardous conditions;
    Resolve positions of all surrounding objects;
    Decide the speed and direction of movement;
}
```

Most real-time systems designed for the robot would try to figure out the worst case execution time for all the steps in the loop and make sure they can finish within the time available. During run-time, the system scheduler may interrupt the loop execution if the deadline is reached in the middle of any iteration. When a computation is interrupted before its completion, usually no result can be produced. To avoid these situations, most traditional systems adopt the pessimistic approach of ensuring that even in the worst-case, the loop will complete within its periodicity deadline. The program is
Moreover, such programs are still vulnerable to hardware failures.

To ensure timing constraints are always met, two measures are often used: scheduling algorithms and exception handling. Most scheduling algorithms deal with only simplified or limited numbers of conditions [24]. For example, algorithms have been designed for a set of periodic tasks on one or multiple processors [5,17], for a set of sporadic jobs with precedence relation on one processor [3], etc. Most scheduling algorithms require the execution time of each job to be fixed. However, the execution time of a computation may be difficult to determine for many reasons. For example, the communication delay in distributed systems may not be deterministic. Dynamic workloads in many systems may cause the execution time to fluctuate. Certain program logic (e.g., while loop) requires variable execution time depending on the data value processed. As a result, a scheduler using a simple scheduling algorithm may be crippled by an unexpected lengthy job. Exception handling mechanisms have been suggested to respond to abnormal timing conditions. However, it is impossible to provide exception handlers that can react gracefully if there is simply no time for any reaction. That is, exception handling may be used to abort or undo a computation, but not necessarily to satisfy a timing constraint.

In addition to utilizing scheduling algorithms and exception handling, another possible solution is to design programs that may use different amounts of time or resources to produce results of different qualities. This is accomplished in FLEX using the model of imprecise computation. An imprecise computation may adjust the precision of its result according to the time and resources available. For instance, the function in a robot that interprets vision signals may provide only the rough positions of surrounding objects. To achieve a high responsiveness from the system, imprecise computations are designed to be monotonic so that it is always ready to present the best result produced. We discuss the FLEX constructs for imprecise computation in Section 4.3.

3. Constraint mechanism in FLEX

Programs in FLEX consist of a set of objects. Each object defines statics which store values and procedures which access and modify the statics. The statics carry state information and the procedures cause state transformations. An invocation of the procedure "op" in the object "server" is written as

\[
\text{result} := \text{server.op}(\text{param});
\]

Procedures may invoke other procedures either in the same or in a different object. Procedures may also invoke functions which are stateless. Functions and procedures have local variables. Locals and statics are generically referred items.

In FLEX, constraints are used to express relationships among items. The relationship must be established and maintained throughout the scope of the constraint. The scope of a constraint is a statement or a group of statements, called a constraint-block (CB), that follow it. Constraint structures can be nested, i.e. constraints can be defined inside of a CB. As an option, exception handling may be specified for situations when the constraint cannot be satisfied. The symbol "\(\Rightarrow\)" is used to denote the exception handler. A constraint has the following syntax:

\[
\text{[label:]} \text{ bool esp } \text{[\text{\(\Rightarrow\) exception_handler:}]} \{ \text{CB} \}
\]

For example, the following statement

\[
C1: \text{distance} < 1 \Rightarrow \text{react}(\text{object});
\]

\[
\{
\text{foreach object do}
\begin{align*}
&\text{get_distance}(\text{object});
\end{align*}
\}
\]

causes the loop to check that the robot is safe from each object. The loop constitutes the constraint-block, and C1 is an optional label for the CB. If an object is too close and the constraint is violated, then the exception handler "react" is triggered. The exception handling mechanism supports various models of responses; namely, resume, terminate or retry the CB [28]. We decided to provide all three models since it may or may not be possible to repair the constraint. To achieve the best efficiency, a CB should be resumed or retried as much as possible. However, certain conditions may not be recoverable and the only option for the exception handler is to terminate the CB. This is usually the case for timing constraints to be discussed next.

3.1. Timing constraint

To model the notion of time, FLEX has a system-defined clock object. The object is simply called Clock with two procedures defined: get_time returns a structure containing the current time and delay takes an interval value and returns only when the time interval expires.

The constraint mechanism in FLEX is used to express timing requirements. Several timing attributes are defined for CBs. Each CB has the following attributes defined:

1. **start**: an absolute time at which the execution of the CB begins.
2. **finish**: an absolute time at which the execution finishes.
3. **duration**: a relative time during which the CB is in execution. It is less than or equal to the time interval from start to finish.
4. **period**: a relative time between successive executions of a periodic CB.
5. **priority**: an integer value indicating the importance of the CB.
Before the CB is entered, the value for each attribute is undefined. Once the CB starts, attributes will be updated from time to time to reflect the actual timing events. They are also checked by the constraint conditions to see if there is any violation. After the CB has terminated, each attribute will keep the value which was last assigned. They will keep it until the next time when the CB is executed again.

The only exception is the period attribute which has a value fixed when the CB is entered. A periodic CB (PCB) has the period attribute defined and is executed repeatedly until conditions in the constraint become false. The duration attribute for a PCB refers to the execution time of a single iteration of the PCB. The start, finish and priority attributes refer to the entire lifetime of the PCB. The exception handler is invoked if the duration or period attributes are violated in any iteration of the PCB.

To see how these attributes are used, we will show some examples here.

**Example 1:**

\[
C_2: (finish < (last+10)) \land (priority > 2) \{ ... \}
\]

The most often used constraint is to simply specify the deadline and the priority of some operations. This can be defined as in Example 1 where last is a time variable. The constraint can be interpreted as “the finish time for all operations must be less than last+10, and the priority of the execution must be greater than or equal to 2”. If the deadline is missed or the priority is reduced to less than 2, the constraint is violated and an exception will be raised.

**Example 2:**

\[
C_3: (period==30) \land (radiation<=10) \land shutdown(position); \{ ... \}
\]

To define a periodic job, the period of a CB is specified. During the execution of a periodic job, certain constraints can be specified as in Example 2. The period defined for the PCB is 30. If the period of execution is not satisfied, or the radiation level is higher than 10, an emergency shutdown exception will be triggered.

**Example 3:**

\[
C_5: (duration<(5-J1.duration)) \land (duration<4) \{ ... \}
\]

More complicated constraints can also be defined. In some systems, the deadline or execution time depends on the actual behavior of other CBs. We allow the timing attributes of a finished CB to be used in constraints. In this example, suppose two jobs collectively must be finished in 5 minutes and each job must not take more than 4 minutes. If the first job has used up 3 minutes, then the second job must finish in 2 minutes. This example shows that the constraint refers to the timing attributes of J1. If, for any reason, J1 has not been executed yet, the condition will not be examined since no value is defined for J1.duration. This capability of cross-block reference makes the constraint mechanism very powerful. This also allows the timing mechanism to be totally dynamic since the values will be known only after that CB is executed. To our knowledge, the capability is not provided in any other language design.

**Example 4:**

\[
Clock.get_time < (loop_start+5) \land shutdown(); \{ ... \}
\]

An interesting feature is to make the CB a null statement. This allows a timing constraint to be checked at that particular execution point, and an exception is raised if the constraint is violated. In this example, the constraint can be used to gracefully terminate a loop if too much time has been consumed to reach the statement. In other words, a constraint can be maintained for a period of time (statements) or checked at a particular instant like an if statement.

Thus it is possible to express complex timing requirements. According to [5], constructs for expressing timing constraints of real-time systems should be able to handle specifications of the minimum and maximum interval between events, and of durations of events. The time attributes proposed can handle all that is required. The primitives may have some duplicated capabilities. For instance, the period attribute could be expressed by adding an inner constraint which defines the start time of an iteration in terms of the start time for the previous iteration. However, period is such a common attribute that providing it makes some constraints much easier to understand.

3.2. Resource constraints

Traditionally, the CPU is considered the most important resource in a computer system. However, many other resources are also crucial to the correct execution of a program. A real-time program may not produce a correct result if some I/O or communication facilities it needs are not available to it. In some systems, if a resource is not available, it may be possible to dynamically adjust the computation to work without it. For all these considerations, it is necessary to have a way to express resource requirements and to handle abnormal events.

In FLEX, all resources are defined as objects. A hardware device can be modeled as an object with procedures corresponding to its interfaces. Software objects may be used to simulate or translate the interface of a hardware device. Resource requirements are expressed in FLEX by using the constraints on resource objects. Suppose the memory object defines an avail procedure which returns the amount of free memory, the constraint

\[
Memory.avail>low \rightarrow array_size:=small; \{ ... \}
\]

may be used to check if there is enough memory available. If not, a smaller array size is selected. There are three basic resource categories defined in FLEX: exclusive, shared and synchronized. Exclusive resources (e.g. CPU) can be assigned to only one job at a time. Shared resources (e.g. shared memory)
may be assigned to multiple jobs at the same time. Synchronised resources are defined for resources like synchronised communication in which the two or more parties involved must use the resource at the same time. Since any interface to the environment may be modeled using objects, system dependent compiler implementations may allow any resource constraint to be specified and handled in the above manner. To provide portability, however, FLEX defines some basic resource objects commonly used in programs like Memory, CPU and Communication.

A novel way to use resource constraints is to implement the self-timing capability in the program. The program can measure some dynamic characteristics of resources at run-time before committing to an execution strategy. For example, a dummy message can be exchanged between two sites to test for the communication speed. If the network is slow or heavily loaded, an program path with less communication should be selected. A constraint for such purpose can be implemented like:

\[ \text{Channel.delay <== 5} \rightarrow \text{local_op:=true;} \]

In the CB, the roundtrip delay for communication is recorded. The delay must be less than 5 time units for the normal computation to complete before the deadline. Otherwise, the computation will choose to conduct local operations as much as possible.

In Ada, programmers can use pragma to direct the compiler to perform some semantics-dependent translations. Some predefined pragmas include priority, memory_size, optimize (time or space), and pack. Our resource constraints may be used to implement all options provided by pragmas. However, one major difference between the two mechanisms is that resources are used as first-class objects in FLEX. They can be dynamically referenced and manipulated. Pragmas in Ada are only directives to the compilers. It is not the purpose of the pragma mechanism to provide flexibility at run-time, or to modify the program control logic.

4. Designing monotonic programs in FLEX

Constraints by themselves are not sufficient to provide the correctness in timing required by real-time programs. Some other mechanisms are still required to enforce or maintain the constraints. A common approach to enforcing timing constraints is to adopt scheduling policies which ensure that the desired resources are available to each jobs before its deadline. However, sometimes schedulers may not be able to predict and to enforce the actual timing conditions. In addition to a powerful scheduling mechanism [4,18], we incorporate a complementary approach to maintain the constraints. It is done by designing imprecise computations in constraint blocks so that they may provide acceptable results and still satisfy the timing constraints. In other words, we allow the execution time of a CB to be modified, just like other first-class object, in a prespecified fashion. Unlike other real-time language designs which treat time as a read-only value, our model of execution time provides a high degree of run-time flexibility in responding to all sorts of dynamic conditions. By using both measures, we hope that timing constraints can be maintained to a larger extent.

Imprecise computations are especially useful if they are monotonic [16] in the precision of results produced. In other words, the quality of the result produced should not decrease as more time and resources are consumed. They should reach closer and closer to the final, or precise result which would be produced if the computation ran to completion. If a CB is monotonic, then the scheduler can allow the CB to execute as long as possible knowing that the result will be strictly improved. In this section, we discuss how monotonic computations can be implemented by special primitives in FLEX.

4.1. Refining imprecise values

Many computations involve optimization after an initial feasible solution is achieved. One way to implement monotonic computations is to first arrive at approximate results, and then improve the results as much as time permits. The inputs and outputs of many numerical methods as well as general computations are values with some inaccuracy. FLEX provides ways to directly represent and manipulate imprecise values.

Two kinds of values are used in FLEX. The primitive values in FLEX are values as in other languages, such as integers, real numbers, characters and user-defined constants. In addition, FLEX also manipulates derived values which can be obtained from primitive values. A derived value represents alternative possibilities. For example, the derived value \([0,1]\) represents a primitive value which is either 0 or 1, but cannot be currently identified. A derived value can also be expressed as a range like \([1..7]\). It should be noted that a derived value is not a set of values but represents alternative possibilities for the primary value.

A type in FLEX consists of a set of primitive values and all the derived values which can be constructed from them. It also includes two special derived values; \(T\) and \(L\). \(T\) represents "any primitive value in the type" and \(L\) represents "none of the primitive values in the type". For example, for type Boolean, \(T\) is the value \([true \mid false]\) and \(L\) is the value \([\_]\). Including \(T\) and \(L\) in a type makes the values of the type a lattice partially-ordered by the \(\leq\) or \(\preceq\) relation [8,22]. For example, \([1,2] \leq [12,3]\), and \([1] \leq [12]\.

Definition: Given two values \(m\) and \(n\), \(m \preceq n\) if every possibility included in \(m\) is also included in \(n\), i.e. \(a \in m\) implies \(a \in n\).

Refinement is a special way to modify the value associated with an item. Refining an item \(x\) with another item \(y\) reduces the possible values of \(x\) to those which are...
also possible values of \( y \). For example, if \( x \) is \([1\,2\,3]\) and \( y \) is \([2\,3]\), refining \( x \) with \( y \) causes \( x \) to get the value \([3]\). Refinement of \( x \) with \( y \) is written as \( "x \leftarrow y" \). Refinement monotonically improves the precision of an item. The operation improves one's knowledge of a value, as contrasted to the assignment which erases all previous knowledge about the value. Refinement may be used, for example, in the iterative method for finding the common divisor of a set of numbers. Initially, the common divisor is defined to be \( \top \). With refinement, we gradually reduce the divisor to include factors of all numbers tested so far. In this way, the monotonicity of the computation is achieved.

4.2. Enhancing imprecise functions

Similar to the lattice of values, a group of functions can be structured into a lattice. Given the same input \( x \), suppose a function \( f \) produces an output \( y \), and another function \( f' \) produces \( y' \) as output. We define the mapping \( S \) as follows:

**Definition:** Given \( f(x)=y \) and \( f'(x)=y' \), \( f' \in S f \) if and only if \( y' \in R \) for all legal inputs \( x \).

We also define functions \( f\top \) and \( f\bot \), which are functions always producing \( \top \) and \( \bot \) respectively regardless of input values. Functions with the same or similar specification can be structured in a lattice partial-ordered by \( S \) which compares and defines which function is "better" in the sense of producing more precise results.

Monotonicity characterizes a single function in terms of whether it propagates an increase in precision. In other words, a function is monotonic if more precise inputs always result in more precise outputs. Many commonly used operations are monotonic: the basic arithmetic and boolean operations, as well as the character and string operations, and the set operations (union and intersection). Many functions can be expressed as the compositions of these monotonic operations. It can be shown that any function which invokes only monotonic operations or functions will also be monotonic [20].

To improve the result produced, a computation may invoke functions to refine a known imprecise result. We call such functions as **sieves**. A sieve function produces outputs which are at least as precise as the corresponding inputs. Thus sieves provide the tradeoff we have been looking for: to get faster results, we can omit some sieves and receive less precise results earlier. In [20], we have proved that if an invocation to a sieve is performed at the end of a function, the resulting function will always be more precise; while if the sieve is added in the middle, the resulting function is guaranteed to be more precise if all subsequent operations are monotonic. Adding sieve invocations to obtain more precise results is referred as the **enhancement**. Enhancement allows programs in FLEX to monotonically return more precise results when more time is expended.

5. Implementation issues

There are two unique features which distinguish FLEX from other languages from the implementation point of view: imprecision and constraints. We have completed an initial implementation of FLEX. It supports the primitives for imprecise computation, constraints on items and general time constraints. In this section, we discuss some implementation issues involved and also present the performance data.

5.1. Implementing constraints

The most critical issue in implementing constraints is how often and how efficiently the constraints are checked. Constraints are implemented differently for items and time. Constraints involving items are checked once when first encountered, and again whenever any of the items are modified. Code is inserted after these statements to re-evaluate the constraint. When constraints are nested, modification of an item triggers all constraints involving it, with the innermost being evaluated first.

Timing constraints are handled using system timers, to avoid re-evaluation at every clock tick. If timing attributes specify the earliest time to begin an activity, execution is suspended until the timer goes off. If the constraint specifies a latest time to complete an activity, an unfinished activity will be terminated by the timer interrupt, and the exception handler is invoked. If an activity is due to begin but is not yet ready to execute, the exception handler is invoked instead. Event-based values (like \( \text{C1.start} \)) are detected by the compiler, which ensures that the program sets these attributes to appropriate values when they become available.

5.2. Implementing imprecision

Imprecise values may be either a discrete set of possibilities, such as \([1\,3\,8]\), or a continuous range, such as \([1..5]\). FLEX treats every item as being imprecise. The actual value may be either a primitive or a derived value. Since there is no upper bound on the number of possibilities, each item is implemented as a list of values. The major consideration for implementation is efficiency because the decisions affect every single computation performed.

Since the list of possibilities can contain both ranges and primitives, the first issue is whether the list should contain homogeneous or heterogeneous entries. The alternative is to view a possibility such as \( 5 \) to be the range \( 5..5 \). This requires performing each operation twice, but distinguishing between heterogeneous element types involves more overhead. We treat single possibilities as ranges, and when it is necessary to distinguish them, such as for output, we check if the end-points of the range are equal.

\( \top \) and \( \bot \) are implemented using separate fields with special checkings. Primitive values can also be treated as special cases. Since they are so common, it is preferable
to handle primitive values without the overhead of list processing. We share the marker field used for | and , to indicate that the value is a primitive value, and handle them also as special cases.

A possible solution to the problem of efficient handling of primitive values is to add more intelligence in the compiler. The value associated with an item is primitive unless an imprecise value is assigned to it, using imprecision operators, or by assignment from another imprecise value. In many cases, the compiler can detect that a value held is precise so that it is represented and manipulated efficiently. The value can still be converted to an imprecise value when necessary. While this technique may miss some items whose value is primitive, it will catch some of the most common simple cases, such as loop counters.

Performing computations on ranges involves interval arithmetic, which has been the subject of much study [19,21,23]. While for most common operations such as addition it is sufficient to perform the computation on the end points of the range, for other operations more complex techniques are necessary. For applications involving extensive manipulations on intervals, it is best to utilize special hardware to perform interval arithmetic. Currently, imprecise functions (sieves) are implemented through run-time decision statements. The compiler estimates the time for the sieve execution as well as the rest of the CB. A statement is inserted just before the sieve invocation, which checks the remaining time available for executing the constraint. If there is enough time to execute the sieve as well as the rest of the CB, the sieve is executed; otherwise it is skipped. In general, the estimated execution time provided by the compiler will be imprecise. This is not a serious problem because a poor estimate will result only in a loss of precision but not a failure.

5.3. Implementation status

For the current implementation we enhance a C++ translator (which produces C code). Therefore it supports all C++ constructs, along with the FLEX primitives: constraints and imprecision. Constraints are implemented using static analysis and interrupts. The constraint is parsed to provide a list of the items involved. Whenever a statement modifies any of these items, the constraint is checked. The translator extracts timing requirements and inserts delay calls or calls to the alarm function. Since there may be multiple alarms requested, it builds a list of time_to_alarm at run-time. If a CB terminates normally, its alarm entry is deleted from the list. This technique for implementing constraints involves relatively little overhead.

Imprecise values are implemented as new classes imp_int, imp_float etc., and all the arithmetic operations are overloaded. The linked list implementation described above is used, with the pointer to the list aliased (using union) to double as a data value, thus providing a cheap implementation of primitive values. The original C++ translator is modified to recognize the FLEX operators like , < and >.

A special efficiency problem is that of space allocation. Since each element of the linked list needs to be allocated dynamically, and arithmetic operations must return temporaries for their results, (involves copying the entire data structure, allocating each node separately) would involve a lot of overhead. Therefore we added a special-purpose allocator which allocates large chunks from the system, and performs its own space management. The main purpose of our current implementation is to show the feasibility of providing constraint mechanisms and imprecise values. As a result, the code generated by the compiler is rather primitive and inefficient. We are investigating various forms of optimization which, we hope, will make FLEX a useful real-time language.

Table I compares the performance data of FLEX and C++. The first four programs show the average speed of some basic operations. The poor performance of FLEX is due to overhead for copying the list structure when it is passed as arguments and returned as temporary results. The fourth program compares performance for an imprecise multiplication, with the C++ version using standard mathematical techniques for error estimation. The answers obtained are almost identical. The FLEX program is slower, but the performance differential is not quite so great.

The fifth program is for an application for which writing the program in FLEX is comparatively easier than in C++. The programs compare unsorted lists and produce the common entries. A C++ program to do this takes lists of keys and compares them with each other, discarding unmatched keys. The FLEX program obtains an imprecise value (multiple possibilities for key values which match) from each relation. Refinement is used to extract common values. This is considerably easier to program in FLEX, since no list comparison (or sorting) is involved.

<table>
<thead>
<tr>
<th>Programs</th>
<th>C++</th>
<th>FLEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer assignment(µs)</td>
<td>0.331</td>
<td>2.552</td>
</tr>
<tr>
<td>Integer addition(µs)</td>
<td>0.668</td>
<td>38.97</td>
</tr>
<tr>
<td>Range addition(µs)</td>
<td></td>
<td>96.80</td>
</tr>
<tr>
<td>Multiplication(µs)</td>
<td>49.339</td>
<td>188.035</td>
</tr>
<tr>
<td>Merging unsorted lists(µs)</td>
<td>315.4</td>
<td>512.2</td>
</tr>
<tr>
<td>Merging sorted lists(µs)</td>
<td>327.0</td>
<td>112.8</td>
</tr>
</tbody>
</table>
The last program is a variation of the above: the list of keys to be compared are partially sorted. The FLEX program (shown in Appendix) operates on ranges, rather than single values (or possibly several ranges and values interspersed). This simplifies the comparison, hence the FLEX program is considerably faster than the C++ version. However, for applications which FLEX is particularly well-suited, a speedup can be achieved. More importantly, these comparisons do not reflect the real benefits of FLEX, such as simplicity of programs, improved dynamic error checking, well-defined imprecision and the usefulness of imprecise values for fault–tolerance and real-time programming. In particular, no comparison of the constraint mechanism and resource flexibility is possible.

6. Conclusion

Two unique features in the real-time language FLEX have been presented. The first one is the ability to specify resource requirements and timing events using constraints, just like the constraints on any first-class object. The other is the ability to refine imprecise values and to define imprecise computations. Incorporating imprecision in the language level ensures the semantics of these operation better defined. It is also easier to write programs whose execution time can be adjusted against the precision of the results. Many real-time environments are dynamic since there may be load variations and unexpected failures. Using FLEX, the burden of meeting dynamic deadlines is no longer on the system scheduler alone, but also shared by the programmer and the runtime system.

References

Appendix. Program for Merging Lists

In this Appendix we show the programs used in test case 6 of Table I. Several lists, some of which are sorted on the key field being returned and some are not, are merged to produce a list of the common elements. We show first the common global declarations used in both C++ version and FLEX version. The FLEX program is shown next, and then C++ program. The C++ version uses linked lists instead of imprecise values.

```c
#include "imp.h" /* imp_value library */

const Nlists := 4,
LSize := 200;
short Sorted[Nlists], LVal[Nlists][LSize];
struct node {
    int Val;
    struct node *next;
};

/* set up lists using a random number generator */
void createlists()
{
    short i,j,n;

    random(1);
    for (n:=0; n<Nlists; n++) {
        Sorted[n] := (random() % 350) < 50;
        if (Sorted[n]) /* produce sorted values */
            for (i:=0; i<Listsize; i++)
                ListValues[n][i] := i+k+1;
        else /* produce arbitrary values */
            for (j:=0; j<Listsize; j++)
                ListValues[n][j] := (random()%1000)+1;
    }

main() /* FLEX Version: */
{
    int getkeys(int), List;

    createlists();
    List := getkeys(0); /* first list */
    for (int i:=1; i<Nlists; i++)
        List <- getkeys(i); /* merge lists */
}
```
int getkeys(int n) /* return imprecise value keys */
{
    int head;
    short i;

    if (Sorted[n]) /* use range */
        head := LVal[n][0] .. LVal[n][LSize-1];
    else /* use possibilities */
        for (i=0; i<LSize; i++)
            head := LVal[n][i];

    return head;
}

main() /* C++ Version: */
{
    struct node getkeys(int), List[Nlists], *r, **p, **q;
    short found;
    createlists();
    List[0] = getkeys(0);
    for (int i=1; i<Nlists; i++)
    {
        List[i] = getkeys(i);
        p = &List[0].next; /* all lists compared with first */
        while (*p)
        {
            found = 0; /* flag to indicate match */
            for (q=&List[i].next; *q; q=&(*q)->next)
                if (!(found)
                    { r=*p; *p = (*p)->next; delete r; }
                else p = &(*p)->next; /* retain only matched items */
            for (r=List[i].next; r; r=r->next) delete r; /* delete list when done */
        }
    }
}

struct node getkeys(int n) /* return a linked list of keys */
{
    struct node head, **p;
    short i=0;
    p = &head.next;
    for (i=0; i<Listsize; i++)
    {
        *p = new struct node;
        (*p)->val = ListValues[n][i];
        p = &(*p)->next;
    }
    *p = NULL;
    return head;
}