Chapter 14

Issues on Real-Time Systems Programming: Language, Compiler, and Object Orientation

Kwei-Jay Lin

This chapter investigates how real-time system programs can be implemented. We first review the important issues in a real-time programming language. The timing constraint constructs in various programming languages, such as *Flex*, RTC++, and RT-Euclid, are introduced and compared. We also present an implementation scheme of the timing constraint block in *Flex*. The interactions between constraints and the consistency checking mechanism are discussed. In some cases, compilers can be used be improve the real-time quality of programs; we discuss two such proposals. Finally, we show how object-orientation may be desirable in real-time systems, as it may enhance the flexibility, predictability, and schedulability of systems.

14.1 Introduction

Computations in real-time systems [24] have stringent timing constraints. Different from non-real-time computations, a real-time computation must satisfy both the *logical* and the *temporal* correctness criteria; failure to satisfy either makes a result unacceptable. The logical correctness concerns the values of the output generated by the computation and the internal state of the system after the computation is performed. The temporal correctness decides if a computation meets its *absolute* timing constraints, like the *ready time* and the *deadline*, and also its *relative* timing constraints, like the temporal *distance* between computations [12]. The absolute timing constraint defines the temporal relationship between a computation and real-world event, while the relative timing constraint defines the temporal relationship between a computation and other computations in the system. For a *soft* real-time computation, a slight delay in meeting the deadline will degrade the usefulness of

a result; while for a *hard* real-time computation, any result produced after the deadline is considered useless or even counter-productive. For safety-critical applications, a missed deadline may even cause a system or its host environment to have a failure or a disaster. Therefore, it is desirable to provide some forms of *guarantee* on the system's capability to complete critical jobs .

Computations in real-time systems are usually triggered by both external events and interval timers. The program specification [8] includes the times at, before, or after which events and responses may occur, as well as the minimum and maximum time intervals that may elapse between events. To ensure that a program meets its specifications, a real-time programming language must allow

- programmers to express different types of timing constraints,
- compilers to check the feasibility of meeting the timing requirements,
- systems to enforce timing constraints either before or at run time.

Given a program coded and the timing constraints on it, the system must know how much time and resources the program needs in order to check if the timing constraints can be satisfied. The timing problem is non-trivial since many factors may affect the execution time of a program. For example, many processors have operation pipelines which have different execution times depending on the number of branch operations executed. Factors like data-dependent execution path make compile-time analysis impossible. In fact, in the general case, the problem of determining the program execution time is equivalent to the halting problem, which means that we may not be able to tell if a program execution will terminate. With a limited set of language and system primitives which have well-defined temporal semantics (like time-bounded loops or IPC with time-out parameters), however, one may be able to bound the execution time of a program. It is thus the goals of many research projects to design such a limited set of high-level programming constructs so that programmers can better manage the temporal behavior of their programs. With these constructs and timing constraints, intelligent compilers and powerful tools can be implemented to improve the predictability and the schedulability of real-time programs.

Even with a good set of programming constructs, a real-time program can still miss its timing constraints if the amount of computing resources available to it is insufficient. The available resource may change in systems with a dynamically changing task population or unpredictable hardware failures. For systems which must handle dynamic events, tasks may be initiated or discontinued as situations require. For systems with fault-tolerant capabilities and expected faults, the available resources in a system may change from time to time. In many applications, the program implemented must ensure that timing constraints are still met in such situations.

One possible approach to handle the above dynamic requirement is to design real-time computations with a flexible execution requirement (in terms of time and resources needed) and to implement run-time systems which can adjust the time and resources allocated to each computation so that the timing constraints of

critical tasks (i.e., a subset of the tasks that are critical to the system's mission) are met. Such a flexibility can be accomplished by structuring computations as *real-time objects* and by using scheduling models such as *imprecise computation* and *performance polymorphism* to produce the best result under all situations.

In this chapter, we investigate real-time systems programming and related issues. We first discuss the design and implementation of real-time languages like *Flex*, Real-Time Euclid, RTC++, Dicon, and so on. We describe how different types of timing constraints may be expressed, and how they may be compiled or enforced by the compiler and the run-time system. We then review the object-oriented model for real-time systems, pointing out how object-orientation may be used to enhance the flexibility of real-time systems. Our objective in this chapter is to present a snapshot of the state-of-the-art on real-time systems programming. By comparing different proposed approaches, we hope to identify a promising approach in implementing future real-time systems.

14.2 Real-Time Language Issues

In the past, a real-time system programmer's job is to write a program with an execution time which is shorter than the time allowed in the timing constraints. Since it is difficult to predict the precise execution time of a program, it usually takes many rounds of trials and errors to build a dependable real-time system. Moreover, it takes the same level of effort to port existing real-time software to a new configuration than to build it from scratch. This is definitely undesirable since the demands for real-time software are stronger than ever, due to the ubiquity of computers in all application areas. To make matters worse, most next-generation real-time applications have very strict timing constraints and must provide very dependable services. We thus need new real-time programming languages and methodologies so that we can meet these demands.

A possible approach is not to expect a precise execution time from a computation, but to make the system so flexible that it can meet a wide range of timing constraints under different system configurations. Like a good manager, a computation can be designed to make the best use of whatever time and resources are available to it. The system scheduler decides the amount of time and resources a computation can have, and the computation then produces a result using only the time and resources allowed. In this way, the computation is guaranteed to be always *temporally* correct, although it may sacrifice its *functional* correctness by producing less desirable results. In many hard real-time systems, the requirement for functional correctness is not as strict as the requirement for temporal correctness. For these applications, the flexible performance approach is thus preferred.

To develop the next generation real-time systems using this approach, we need to have a programming language which has constructs for directly expressing timing constraints. The programmer defines the temporal constraints for (part of) the computation and provides the possible set of codes to be executed. The run-time system, together with the compiler and the system scheduler, must decide and/or generate the code to be executed so that the constraints are met.

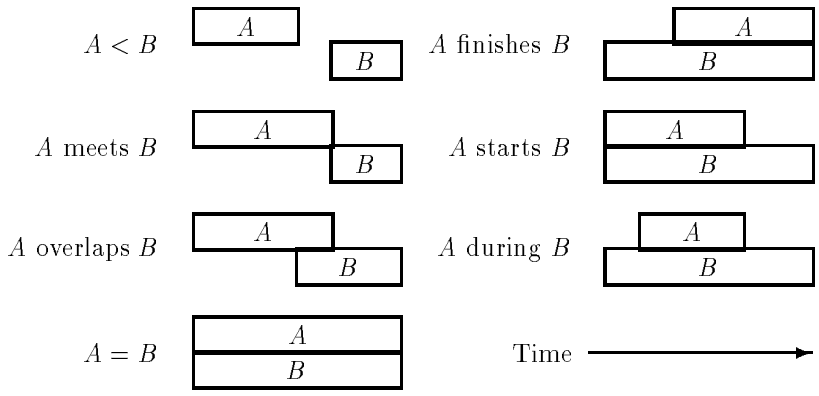

Figure 14.1 Temporal Relations between Computations

Many language features are desirable for real-time systems programming. In fact, almost all features desirable in a "good" conventional (i.e., non-real-time) programming language can be considered as essential for real-time systems programming. For example, five requirements have been suggested for real-time software in [26]: predictability, reliability, tasking, modularity, and maintainability. Of the five, only the predictability is a less familiar requirement for conventional software. The other four are highly desirable in all types of software systems. Although real-time systems may bring new meanings to these four requirements, we will not discuss them in this chapter, due to the space constraint. We will focus only on the predictability issue in this chapter.

In this section, we investigate the real-time constraint issues, including the theory, the construct, and the compilation.

14.2.1 Specification of Real-Time Constraints

A consistent theory of timing constraints has been presented by Allen [1] in his study of how to maintain knowledge about the temporal ordering of a set of actions using the temporal logic. He represents each action as taking place in an interval of time, and defines seven relations between intervals (see Figure 14.1) that describe the synchronization of one event with respect to another. These relations correspond to intuitive notions such as "A takes place before B" ($A < B$), "A and B begin and end at the same time" ($A = B$), and "A takes place during the time that B does" (A during B). Although many other temporal models have been proposed by others, Allen's work provides a good foundation for our discussion of real-time constraints.

One of the first researchers to describe timing constraints in real-time systems was Dasarathy [8]. He constructed a language, called the Real-Time Requirements Language (RTRL), for the event-action model. In his scheme, timing constraints could express the minimum or maximum time allowed between the occurrence of stimuli S, actions in the outside world, and responses R, the completion of the

actions that a system takes. All four combinations $S-S$, $S-R$, $R-S$, and $R-R$ could be specified. Constraints on the time before a stimulus were constraints on the behavior of the outside world (e.g., a telephone user in Dasarathy's paper). Constraints on the time before a response were interpreted as constraints on the amount of time that the system could use to process the corresponding stimuli. Note that Dasarathy's RTRL was a specification language, and not intended for automatic processing.

A similar scheme was Jahanian's Real-Time Logic (RTL) [14]. In this scheme events and actions, corresponding roughly to Dasarathy's stimuli and responses, were identified. A mechanized inference procedure was presented to perform automatic reasoning about timing properties, and the events and responses could be associated by annotation with program constructs. A run-time monitoring system for the constraints specified in RTL has also been implemented, so that the system can adapt to a changing environment or an exception condition [7]. Like RTRL, RTL is designed to be a specification language.

14.2.2 Real-Time Constraint Constructs

The executions of real-time programs must meet timing constraints. As mentioned earlier, many different types of constraints may be required of a real-time computation, including *absolute* (e.g., deadline and duration) and *relative* (e.g., distance and frequency) constraints. We believe that these constraints should be explicitly defined in the program, rather than implicitly embedded in the code. This is because an implicit timing constraint may not be guaranteed whenever the normal execution flow is interrupted by unexpected events or altered by new system configurations. It is better to have timing constraints explicitly defined so that it is known by the system scheduler and other system resource managers (like network manager and I/O device manager).

Real-Time Euclid is a language extended from Euclid with real-time constructs and with provisions for "schedulability analysis," i.e. to verify that software adheres to its timing constraints at compile time. The effort, in fact, is to make real-time software as predictable as possible since it is impossible to analyze the schedulability without taking into account the actual system configuration and run-time support (like OS support, scheduling model, network protocol, etc.). In other words, some part of the schedulability analysis may need to be performed outside the compile time. In Real-Time Euclid, timing constraints are defined by the frames associated with processes. Each process must complete its task before the end of the current frame, and cannot be reactivated until the end of the current frame. To be able to verify that software can meet its timing constraints, Real-Time Euclid has no constructs that can take arbitrarily long to execute. For example, loops must have constant counts. No recursion and dynamic variables are allowed. Wait- and device-condition variables, as well as exception handlers, are all time-bounded.

RTC++ [13] is an extension of C++ with the real-time object model. In RTC++, active objects can be defined to have timing constraints like the worst-case execution time *(Bound)* and the deadline *(Within)*. Periodic tasks can also be defined by the *cycle* statement, which has the parameters for *start time, end*

time, period, and *deadline*. For schedulability analysis, RTC++ adopts the rate monotonic scheduling analysis as the scheduling model. It also adopts the priority inheritance protocol to control the priority inversion problems when concurrent threads must share resources. A similar work is the Real-Time Concurrent C [2], which is an extension of the Concurrent C language. Real-time constraints such as deadline and periodicity can be defined. Another construct called *guarantee* is provided so that the run-time system will check if the statement can be guaranteed to complete before the deadline. If not, an alternate computation is executed.

More complex definition of timing constraints have been investigated by other projects. One early work is Dicon [17], which defines the *temporal scope*. Dicon is a distributed configuration language in which for each task, one can specify the deadline, the minimum and maximum delay before execution begins, the maximum execution time, and the maximum elapsed time. This is the first language where one can constrain both the upper and lower bounds of some temporal behavior.

Another timing constraint proposal which can define not only the upper and lower bounds, but also the absolute and relative constraints is the *Flex* language [15, 19], which is also an extension of C++. *Flex* reasons about time and resources by specifying *constraints* and propagating information among them. In *Flex*, constraints on time and resources are described by the *constraint block.* A constraint block identifies a constraint that must apply while a section of code is in execution. A constraint may be either a Boolean expression (which is treated as an assertion to be maintained throughout the block's lifetime) or a timing constraint, which describes a constraint on the time at which the block may begin or end its execution.

Associated with each constraint block is an interval of time representing the lifetime of the block. Another block may refer to the *start* and *finish* times of a given block by using the block's label; thus $A.start$ represents the start time of block A, and $A.finish$ represents the finish time of A. The boundaries of a constraint block may be designated in terms of the relative time from the start of the block. $A.duration$ represents the difference between the *start* and *finish* times of A, and $A.interval$ represents the difference between the *start* time of one execution of block A and the *start* of the next time it is executed. Figure 14.2 shows all four of these timing attributes.

Timing constraints in *Flex* take one of the forms shown in Table 14.1. They can be classified by the following properties:

Start or Finish Time. Constraints can refer either to the time that the computations represented by a constraint block begin, or to the time that they complete.

Absolute or Relative Time. Absolute times represent the actual time of the start or finish event. Relative times represent the elapsed time from the start event to the event described.

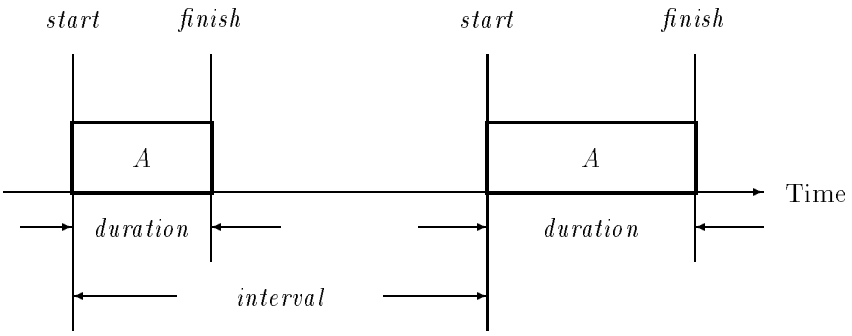

Figure 14.2 Timing Attributes of Constraint Blocks

Table 14.1 Timing Constraints in *Flex*

	Absolute Time		Relative Time	
	Start	Finish	Start	Finish
Earliest	$start \geq t$	$finish \geq t$	$interval \geq t$	$duration \geq t$
Latest	$start \leq t$	$finish \leq t$	$interval \leq t$	$duration \leq t$

Earliest or Latest Time. Constraints can refer either to the earliest time at which an event may occur, or to the latest time at which it is permitted to occur.

The left-hand side of a timing constraint must refer to a timing attribute of the current block. The relational operator ($\geq$ or $\leq$) can be used to specify either the earliest or the latest time for executing the block, or the upper and the lower bounds on event distances . The right-hand side of a constraint can be a constant, or an expression involving the timing attribute of another constraint block. Thus a timing constraint on a constraint block can be defined *relative* to the execution of another block, not just to the global clock. When one block refers to the attribute of another block that may be executed many times (like periodic tasks), the values always refer to the most recent activation of the block.

With the timing constructs provided in *Flex*, one can implement all the timing relations as defined in Figure 14.1. One can also easily define temporal constraint macros for *periodic* and *aperiodic* computations.

14.2.3 Checking Real-Time Constraints

Given the timing constraint constructs defined in a real-time language, the compiler must generate the code to enforce them. In this section, we present how the real-time constructs in *Flex* are implemented since it has one of the more complex constructs.

There are three basic temporal data types in *Flex*: `Tick` , `TickInterval`, and

```
Episode E;       // Episode describing block's execution time
{
  Context __C_n;              // Establish context
  if (__C_n.save())
  {
    -- user's exception handler goes here --
  }
  else
  {
    -- declarations for non-temporal constraints --
    -- declarations for latest finish time --
    -- declarations for latest start time --
    -- code to delay until earliest start time --
    -- code to clean up latest start time structures --
    E.start = NOW ();     // Record the start time

    -- body of the constraint block --

    -- code to delay until earliest finish time --
    -- code to clean up latest finish time structures. --
    -- code to clean up non-temporal constraints --
   }
 E.finish = NOW (); // Record the finish time
}
```

Figure 14.3 Structure of a Constraint Block

`Episode`. In *Flex*, time is represented as a set of discrete, quantized instants. Each instant is represented as an instance of the data type `Tick`. There is a function called `NOW()` that returns the current absolute time. A `Tick` is a precisely known instant of time. In many cases, such as an event that is to occur in the future, the time of an event is not known precisely. The `TickInterval` type represents these uncertain ranges of time. The `Episode` data type represents the period of time during which some constraint block is executed. It comprises two attributes: the *start* time, at which the process begins, and the *finish* time, at which the process is completed. Both of these times may be uncertain, and are therefore expressed as intervals. In addition, there are two values that may be determined for an activation: *interval,* the period of time between successive starts, and *duration,* the period of time between a start and a finish. With these temporal data types, the basic structure for the compiled *Flex* constraint block [20] is as shown in Figure 14.3.

Given a real-time program implemented with a set of timing constraint blocks, the system must be able to detect any constraint inconsistency. For example, two blocks may both specify that it has to be executed before the other. Some of the inconsistencies can be checked at the compile time, while others can be checked

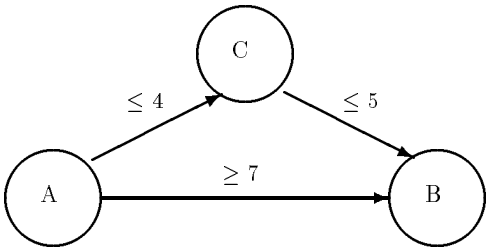

Figure 14.4 A Simple Program Graph

only at run time. Simple checking can be performed for the consistency between *duration* and (*finish* − *start*). For example, if *duration* is defined to be ≥ 10, but (*finish* − *start*) can only be ≤ 8, there is an inconsistency.

Given a set of constraint blocks with distance constraints [11] such as

$$A: \ (start \geq B.start + 5)$$

or

$$A: \ (start \leq B.start + 5)$$

one can check their distance consistency by propagating the relationship in a program graph like Figure 14.4. In a program graph, each constraint block is represented as a node. Each '$A \leq B$' constraint defines a '$\leq$' edge from A to B and each '$A \geq B$' constraint defines a '$\geq$' edge from B to A. The constant in the constraint defines the value of the edge. Algorithms have been devised to check the consistency in the graph [20].

14.3 Meeting Timing Constraints

Given a set of timing constraints defined in a program, the compiler and the run-time system must provide the means to meet them. One approach is to compile the program just like a non-real-time program, and then predict the performance of the computation as implemented. With the performance prediction, the system can check if the constraints can be satisfied in a specific run-time environment using a specific scheduling algorithm. Another approach in meeting the timing constraints is to compile and to transform the program so that the code generated is guaranteed or at least has a better chance to meet the constraints. This again must take into account the run-time scheduling model used.

The problem with the former approach is the performance prediction. One possible way to determine the performance of a program is to run it, either on the target hardware itself, or on some sort of simulator that models the hardware. The simulation or test run must be presented with data that are representative of the data that will be seen in actual service of the system. The problem with this method

is that the testing is limited, and generally cannot include the entire set of possible inputs. The worst case for the consumption of time or some other resource may not be uncovered in testing. Moreover, if a simulator is used, there may be some uncertainty that the simulator actually reflects the performance of the underlying hardware. For this reason, many have proposed conducting analyses that examine some form of the program code and attempt to prove assertions about the program's performance behavior. Using this method, the worst-case performance can always be identified so that any guarantee made is absolute.

In this section, we review some of the work on predicting program performances and on meeting timing constraints.

14.3.1 Predicting Program Performances

An early research to attempt to characterize the performance of high-level programs was conducted by Leinbaugh [18]. This work characterized the time required by a higher-level construct in terms of its CPU time, time spent waiting to enter a critical section, time spent ready and waiting for a processor, time spent performing I/O, and time spent waiting at a synchronization point. Conservative bounds for all of these quantities were estimated.

Stoyenko [25] adopted Leinbaugh's ideas into a programming system for Real-Time Euclid. Among its features was the ability to verify, a priori, that all resource and timing constraints would be satisfied. Rather than using Leinbaugh's conservative time bounds, their system attempted to find tighter bounds on execution time by an exhaustive search of all possible sequences of execution of processes. However, Stoyenko made the analysis easier by forbidding many programming constructs, including **while** loops (only counted loops were allowed), recursion, and recursive data structures such as linked lists. In this way, tight time bounds could be established for all programming constructs, at the expense of increased programming effort in coping with the limitations of working with the set of constructs remaining.

Mok et al. [21] have taken another approach to this *a priori* program analysis. The performance analyzer works on an abstraction (the interval partition) of the flow graph of the program, and the programmer supplies the worst-case number of iterations of each unbounded loop using a separate timing analysis language. Other work on program performance analysis includes the static approach by Puschner [22], who has refined the calculation of time required in **if** ... **then** ... **else** constructs, and Shaw [23], who has presented some rules for formal verification of programs' timing behavior.

Kenny and Lin have presented an alternative approach based on program measurements [16]. It alleviates some of the objections to program measurements by showing how statistical confidence in the program's performance behavior can be achieved. It also allows actual behavior, and not just worst-case performance, of programs to be estimated, by allowing a performance model to contain dependencies on the input data. In their approach, the system is provided with a performance model which defines a task's execution time as a function of its input data. The model can incorporate the programmer's knowledge about the expected timings. The system then collects the performance data measured from the actual run of

the task on some input data. The system then gives a measure of statistical confidence that the model accurately represents the program's actual behavior. Such an approach may be desirable since a system will be able to:

- analyze program structures that are impossible for other systems, such as unbounded loops and recursive control structures,
- provide accurate timing information even on hardware whose timing behavior is difficult to model and analyze, and
- provide confidence in the timing model, which should improve statistically as more experience on a computation is collected.

In summary, the performance prediction for real-time programs (or any type of programs) remains a difficult problem. The static analysis approach can be used to provide an estimate on the worst-case execution time. Given that real-time systems must provide guarantees on meeting timing constraints, the approach is viewed to be safer and more acceptable. However, such estimates are very pessimistic and may be too conservative for most practical applications. Moreover, program execution time often is dependent on the input data and normally is much less than the worst-case execution time. Thus if every program uses the worst-case execution time for system design, the system implemented will be largely under-utilized. Again, we believe that the flexible performance model can help to achieve a better system efficiency. One can identify a subset of the critical computations in a system to perform the worst-case scenario analysis using their worst-case execution times. In addition, one can perform a normal case analysis using the execution-time predictions from the measurement approach. In this way, the system can be designed to have a high system utilization using the normal case schedule and still is guaranteed to perform acceptably under the worst-case scenario.

14.3.2 Improving the Schedulability

In all the work discussed above, the compiler is used to generate real-time codes with very little consideration for performance issues, except to provide some mechanism for checking and detecting timing errors. However, as demonstrated in many parallel programming systems with optimizing compilers, compilers can be used to capture certain high-level knowledge about the program and produce a better code than that produced by simple-minded compilers. Traditionally, compilers have been used to reduce the time or space required to run a program, or to increase the parallelism in a program. For real-time systems, it may be possible to use compilers to improve the quality of real-time programs. We present two such projects in this section. One design is to increase the schedulability of codes. The other is to increase the flexibility of the system.

In [10], a technique called *Compiler-Assisted Adaptive Scheduling* is presented. In this approach, the compiler examines the application program and inserts the measurement code at appropriate boundaries so that computations can be adapted to avoid violating timing constraints. In addition, code reordering is performed

to allow greater adaptability and early failure detection. In the compiler, codes are classified by their predictability and monotonicity. If possible, non-predictable codes are executed before predictable codes, and non-monotonic codes are executed before monotonic codes. This is because the non-predictable code may take an unknown amount of time: If it takes longer than originally estimated, an early execution will given it a better chance to meet the deadline. But if it takes shorter than expected, the time left can still be utilized by other tasks. Similarly, the monotonic code is executed last since they can be terminated at any time, as required.

In [9], compiler techniques are used to move codes from blocks constrained by tight deadlines into blocks with sufficient slack, and also from tasks with a tight deadline to others without. In this approach, the codes needed to produce an observable event are separated from those that do not. The compiler then moves the latter from the original control flow to after the timing constraint is satisfied by the former. The technique is further tied in with the RM scheduling model to enhance the schedulability.

Other approaches to improve the real-time quality of programs with the aid of compilers are possible. This is a promising area for producing schedulable and flexible real-time programs. Several new projects have been initiated, but more effort should be expended in this direction.

14.4 Adopting Object Orientation

Most real-time systems implemented in the past use the traditional process-oriented model where tasks are implemented as functions performing in a global system state. Some recent real-time systems have adopted a new software paradigm called the object-oriented model. In object-oriented systems, all entities in the system are defined as objects. An object may invoke methods defined in itself or in other objects for the services needed, which in turn may invoke methods in other objects. Each object is defined with a specific type. Types may form a hierarchy where some types inherit the definition of other types. Cardelli and Wegner [6] have therefore identified the three basic elements of object orientation as:

$$object\ oriented = data\ abstraction + object\ types + type\ inheritance. \quad (14.1)$$

With object type and type hierarchy, when an object is requested to perform a method, it must *bind* the request with the definition of a method which may be defined outside the object in the type hierarchy. Sometimes, the same method requested may be bound to different methods according to the data it needs to process, using a mechanism called *polymorphism*.

Lately, the object-oriented model has received much favorable advocacy from the software community. One of the reasons is that the object-oriented model allows one to apply the principles of hierarchical structuring and component abstraction, which are essential in building large systems. The object-oriented approach also promotes component reusability (by type inheritance), which makes systems easier to maintain and to modify.

For real-time applications, the above are very important benefits. In addition, there are many other good reasons why object orientation is attractive. Most real-time systems have physical devices which can be modeled naturally as objects. The object model may allow object designers to address the timing and performance issues in each object directly. Therefore, if designed correctly, it may be easier to port a real-time object from one configuration to another. It is also easier to optimize objects independently to improve their performances. Finally, objects can be enhanced with different capabilities using the type hierarchy, with different types corresponding to different generations of a real-time system component.

With all of the benefits, however, many believe that object orientation is counter-productive since there may be extra heavy overheads in dynamic binding and dynamic memory management. However, the overheads in these mechanisms can be minimized with a careful compiler and run-time implementation. Moreover, object-oriented real-time systems may provide more flexibility, which may result in even higher performance gains than those attained by non-object-oriented systems. To meet the strict requirement of graceful system degradation in safety-critical real-time systems, object orientation may enhance the predictability of system performance by using a flexible scheduling framework, and by integrating the performance consideration into the structures of object types and the whole system. We will discuss these benefits in this section.

In this section we investigate the issues of implementing object-oriented real-time systems. We review three such proposals: namely, RTC++, Chaos, and *Flex*. Although all of them intend to enhance the adaptability and flexibility of real-time objects, each of the languages adopts object orientation for slightly different reasons.

14.4.1 Real-Time Objects in RTC++

The motivation for RTC++ is to extend C++ with some real-time constructs so that programmers can use the popular language to design real-time programs. To achieve that, the notion of real-time objects have been defined. Some of the issues considered important for real-time programming include (1) timing specification, (2) priority inversion, and (3) multiple threads.

Real-time objects in RTC++ are defined to be active objects with timing constraints. The design of RTC++ places the emphasis on the schedulability of real-time objects using the rate monotonic fixed-priority scheduling model. All considerations for real-time objects are based on the scheduling theory; i.e., the language provides enough timing constructs to programmers so that the system can support the RM scheduler. In other words, objects in RTC++ do not have more capability than objects in C++, other than RM scheduling.

14.4.2 Dynamic Objects in Chaos

In some real-time applications, the system must be able to handle the dynamic nature of the real world to provide a dependable service. For example, avionic software must be able to handle unexpected weather conditions and bursty air

traffic. In such applications, real-time software must be able to adapt to changes in the target environment and yet behave in a predictable manner.

Chaos is an object-oriented language and programming/execution system designed for dynamic real-time systems. The designers of Chaos have chosen to use object orientation in real-time software for the following reasons [3]:

- Using the process-oriented model, the problem decomposition must divide the system according to the functional aspects of the system. In other words, the mapping from software modules to real-world entities, and vice versa, is not apparent. This makes the codes difficult to understand, to maintain, and to reuse.

- The process-oriented model often implements the event-action view: i.e., events are generated by the real world and the software must produce responses. However, the discrete events are actually approximations of continuous activities in the real world. By mapping the activities into pre-defined discrete events, some aspects of the activities are lost and the software loses the flexibility to handle them in different, better ways.

- The object-oriented model is able to represent directly the structural or anatomical aspect of the real world by object definitions. Moreover, since every entity in the system is uniformly represented as objects and communicated by messages, code reuse and maintenance become easier.

Chaos objects and messages have both functional and temporal attributes. Chaos has three components: a C-based run-time library, a programming environment with an entity-relationship database, and a specification language. A programmer specifies the mapping between real-world entities and corresponding software objects. The environment then generates the configuration files, which are then compiled to create object classes and instances. The environment is also used to analyze the application's temporal behavior and to monitor its execution at run time.

Chaos has carefully investigated the predictability issues on dynamic binding, dynamic instance creation, and dynamic class creation mechanisms. Another major effort is the object's ability to negotiate temporal constraints placed on it. In general, Chaos provides very powerful and dynamic mechanisms for object-oriented real-time systems.

14.4.3 Performance Polymorphism in *Flex*

One approach to enhance the flexibility of real-time software so that timing constraints are always satisfied is to change the software structure so that the amount of work performed is based on the amount of time and resources available. In other words, instead of defining a fixed amount of work to be performed, we can define a set of workloads which may or may not be completely executed. During run-time or system reconfiguration, a subset of the workloads is executed using only the amount of time available. The system design and scheduling issue is then to select

the optimal subset of the workloads which gives the best reward under the available time and resources.

In real-time systems, the approach can be implemented in three different ways. First, a computation may actively evaluate its timing constraints to select the execution path with the most desirable response time. Second, the run-time system, given global scheduling knowledge, may bind a real-time request dynamically to a server with appropriate time and resources available. Finally, a computation may resort to producing imprecise results if its timing constraints are so dynamic that they are beyond the control of the previous two mechanisms. The novel feature in this approach is that the execution time is modeled as a first-class object so that it can be modified if necessary.

In object-oriented systems, some methods in an object may be provided by other objects; these methods are bound based on the class hierarchy, and by the parameters of the invocation. In *Flex*, the concept is further generalized to include the execution performance as one of the binding parameters. This binding based on architectural or performance criteria is a form of polymorphism. Instead of having multiple procedures that perform the same action on objects of different types, we can define multiple procedures in *Flex*that perform similar functions based on different environmental constraints.

This model of *performance polymorphism* raises some scheduling issues relating to the binding of the polymorphic operations. When jobs are performance polymorphic computations, jobs do not have a fixed amount of execution time, but can be executed for a variable amount of time. Each version defines a reward function which specifies how much reward can be received for a given execution time. For non-real-time systems, we may want to allocate resources (especially CPU time) evenly to jobs such that all jobs have about the same reward. This is known as the *knapsack sharing* problem, where one maximizes the minimum reward for any job in the system. Brown [5] has proposed an efficient algorithm. For real-time systems the objective is concerned with the allocation of resources to maximize the total value among all reward functions. It is known as the *knapsack* problem [4], which is NP-complete in general. Heuristic algorithms can be used to solve the knapsack problem.

14.5 Conclusions

We have discussed the issues related to real-time systems programming. Three topics have been reviewed: real-time programming language, compiler and tool implementation, and object-oriented real-time systems. In each of the topics, we survey and compare different research proposals and implementations. We show how timing constraints can be monitored and enforced at run time using run-time servers, and discuss how real-time programs can be checked and transformed at compile time. Although much has been done, more refined design and practical experience on the general constructs and programming tools that can be used in a high-level language to enhance its real-time predictability and schedulability are still needed. We believe that such research will provide very helpful results in

building next-generation real-time systems.

Acknowledgments

This research was partially supported by the Office of Naval Research under Grant N00014-94-1-0034 and by the National Science Foundation under Grant CCR-89-11773.

References

[1] James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, November 1983.

[2] T. Bihari and P. Gopinath. Real-time concurrent c: A language for programming dynamic real-time systems. *Real-Time Systems*, 3(4):337–406, 1991.

[3] T. Bihari and P. Gopinath. Object-oriented real-time systems. *IEEE Computer*, 25(12):25–32, December 1992.

[4] J. R. Brown. The knapsack sharing problem. *Operations Research*, 27(2):341–355, March–April 1979.

[5] J. R. Brown. The sharing problem. *Operations Research*, 27(2):324–340, March–April 1979.

[6] Luca Cardelli and Peter Wegner. Understanding types, data abstractions, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[7] S.E. Chodrow, F. Jahanian, and M. Donner. Run-time monitoring of real-time systems. In *Proc. of 12th IEEE Real-Time Systems Symp.*, pages 74–83, 1991.

[8] B. Dasarathy. Timing constraints of real-time systems: Constructs for expressing them, methods for validating them. *IEEE Transactions on Software Engineering*, SE-11(1):80–86, January 1985.

[9] R. Gerber and S. Hong. Semantics-based compiler transformations for enhanced schedulability. In *Proc. of 14th Real-Time Systems Symp.*, 1993.

[10] P. Gopinath and R. Gupta. Applying compiler techniques to scheduling in real-time systems. In *Proc. of 11th IEEE Real-Time Systems Symp.*, pages 247–256, 1990.

[11] C.-C. Han and K. J. Lin. Job scheduling with temporal distance constraints. In *Proc. of 6th IEEE Workshop on Real-Time Operating Systems and Software*, May 1989.

[12] C.-C. Han and K. J. Lin. Scheduling distance-constrained real-time tasks. In *Proc. of the 13th Real-Time Systems Symp.*, pages 300–308, December 1992.

[13] Y. Ishikawa, H. Tokuda, and C.M. Mercer. An object-oriented real-time programming language. *IEEE Computer*, 25(10):66–73, October 1992.

[14] Farnam Jahanian and Aloysius Ka-Lau Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, SE-12(9):890–904, September 1986.

[15] Kevin B. Kenny and K. J. Lin. Building flexible real-time systems using the *Flex* language. *IEEE Computer*, pages 70–78, May 1991.

[16] Kevin B. Kenny and K. J. Lin. Measuring and analyzing the performances of real-time programs. *IEEE Software*, pages 41–49, September 1991.

[17] I. Lee and V. Gehlot. Language constructs for distributed real-time systems. In *Proc. of IEEE Real-Time Systems Symp.*, pages 57–66, December 1985.

[18] Dennis W. Leinbaugh. Guaranteed response times in a hard real-time environment. *IEEE Transactions on Software Engineering*, SE–6(1):85–91, January 1980.

[19] K. J. Lin and S. Natarajan. Expressing and maintaining timing constraints in FLEX. In *Proc. of 9th Real-Time Systems Symp.*, pages 96–105, Huntsville, Ala., December 1988.

[20] K. J. Lin and Kevin B. Kenny. Implementing and checking timing constraints in real-time systems. *Microprocessing and Microprogramming*, 38:477–484, 1993.

[21] A. K. Mok, P. Amerasinghe, M. Chen, and K. Tantisirivat. Evaluating tight execution time bounds of programs by annotations. In *Proc. of 6th IEEE Workshop on Real-Time Operating Systems and Software*, pages 272–279, May 1989.

[22] P. Puschner and Ch. Koza. Calculating the maximum execution time of real–time programs. *Journal of Real–Time Systems*, 1:159–176, 1989.

[23] Alan C. Shaw. Reasoning about time in higher–level language software. *IEEE Transactions on Software Engineering*, 15(7):875–889, July 1989.

[24] John A. Stankovic. Misconceptions about real-time computing. *IEEE Computer*, October 1988.

[25] A. D. Stoyenko. A schedulability analyzer for real-time Euclid. In *Proc. of 8th Real-Time Systems Symp.*, pages 218–227, San Jose, Calif., December 1987.

[26] A. D. Stoyenko and W. A. Halang. High-integrity pearl: A language for industrial real-time applications. *IEEE Software*, 10(4):65–74, July 1993.