Making Software Timing Properties Easier to Inspect and Verify

Jia Xu, *York University*

The world around us is depending more and more on computer systems that have timing requirements. Whether we fly on an airplane, drive a car, make a phone call, undergo surgery at a hospital, or simply turn the lights on at home, we depend on real-time embedded software that must observe timing constraints either for our safety or simply just to make things work. Avionics, air traffic control, automobiles, telecommunications, medical applications such as intensive-care monitoring, nuclear power plants, defense, multimedia, and process control are but a few application areas that employ computer software that synchronizes and coordinates processes and activities with timing constraints. Not only is software with hard timing requirements becoming increasingly important and pervasive, it is also growing rapidly in size and complexity.

In contrast, effective methods and tools for inspecting and verifying software's timing properties are conspicuously absent, despite an increasingly pressing need for them. This is due primarily to the difficulty of verification. However, a *preruntime scheduling* approach can help overcome this difficulty, making software timing properties much easier to inspect and verify.

Software with hard timing requirements should be designed using a systematic approach to make its timing properties easier to inspect and verify. Preruntime scheduling provides such an approach by placing restrictions on software structures to reduce complexity.

The verification problem

What's the main reason for this apparent difficulty in developing effective methods and tools for verifying software timing properties? The problem is the complexity of software, especially nonterminating concurrent software, and the complexity of such software's possible timing behaviors.

Fundamental theoretical limitations

Researchers have found that the ability to express even basic timing properties is a major factor that keeps the complexity of a logical theory or model high. Many of the logics and models that researchers have proposed for modeling programs' timing properties are *undecidable*.[1]

This means that for a particular logic or model, no automatic method or tool can ever exist that always gives a definite answer to the question of whether the program satisfies the timing properties. To make models and logics decidable, researchers can impose fairly severe restrictions such as limiting the system to a finite number of states, requiring the time domain to be discrete, and prohibiting quantification on time variables. However, many logics and models will still have high complexity.

The models and logics that can express basic timing properties are generally subject to the *state space explosion* problem. That is, the state space size we must explore to verify those properties grows exponentially with the program description's size. For example, to verify a program with 200 components, the state space size that we might need to explore might be proportional to 2^{200}! This exceeds the normally available time and memory resources.

To cope with state space explosion, all program verification methods use some form of *approximation* (for more information on current verification methods, see the sidebar). That is, the model only preserves selected characteristics of the implementation while abstracting away complex details. But then we have the problem of deciding how to obtain such a model and how to prove that the model faithfully represents the original program, in the sense that the model can answer correctly all the correctness questions about the program. This can be more difficult than proving the original program's correctness.

State-of-the-Art Verification

Automatic methods or tools based on logics or models that are subject to state space explosion (see the related section in the main article) can still be useful where the program description is small. We can obtain small program descriptions when either the system represented by the program has a simple structure or the number of components is small. Hardware circuits are often regular and hierarchical, and the number of components compared to software is small, so model checking has achieved a fair amount of success in checking the properties of hardware circuits.[1]

However, for the reasons I mentioned earlier, formal verification methods have not been used on the timing properties of actual software code. In particular, large-scale, complex, nonterminating, and concurrent software has a pressing need for this verification. But most research related to such verification has studied only simplified high-level abstractions of software such as specifications, models, algorithms, or protocols that are only approximations of the actual software. These abstractions do not take into account all the implementation details that might affect timing. Examples are theorem-proving techniques that use PVS (Prototype Verification System) to analyze real-time scheduling protocols[2] and symbolic-model-checking techniques that check high-level algorithms and protocols.[3] Most of model checking's success is not so much in the formal verification of specifications but in the finding of bugs that other informal methods such as testing and simulation don't find, through exploring only part of the state space.

Because of the state-space-explosion problem, even state-of-the-art methods and tools have difficulty verifying the timing properties of more than a few real-time processes when the some processes exhibit nondeterministic behavior. For example, the TAXYS tool uses the formal model of timed automata[4] and the KRONOS model checker[5] to verify timing properties of real-time embedded systems. The tool's developers recently reported experimental results in which the tool had to abort when the number of symbolic states that KRONOS explored increased exponentially with the degree of nondeterminism.[6] This increase occurred even though the system being verified contained only two strictly periodic, independent tasks and one aperiodic (asynchronous) task.

References

1. E.M. Clarke et al., "Progress on the State Explosion Problem in Model Checking." *Informatics: 10 Years Back, 10 Years Ahead*, R. Wilhelm, ed., LNCS 2000, Springer-Verlag, 2001, pp. 176–194.
2. B. Dutertre, "Formal Analysis of the Priority Ceiling Protocol," *Proc. 21st Ann. IEEE Real-Time Systems Symp.* (RTSS 2000), IEEE CS Press, 2000, pp. 151–160.
3. S.V. Campos and E.M. Clarke, "The Verus Language: Representing Time Efficiently with BDDs," *Theoretical Computer Science*, vol. 253, no. 1, 17 Feb. 2001, pp. 95–118.
4. R. Alur and D.L. Dill, "A Theory of Timed Automata," *Theoretical Computer Science*, vol. 126, no. 2, 25 Apr. 1994, pp. 183–235.
5. C. Daws et al., "The Tool KRONOS," *Hybrid Systems III: Verification and Control*, LNCS 1066, Springer-Verlag, 1996, pp. 208–219.
6. E. Closse et al., "TAXYS: A Tool for the Development and Verification of Real-Time Embedded Systems," *Proc. 13th Int'l Conf. Computer Aided Verification* (CAV 2001), LNCS 2102, Springer-Verlag, 2001, pp. 391–395.

Complexity's causes

If you take a hard look at what makes the timing behaviors of existing real-time software complex, you might observe the following current practices in the design of real-time software:

- The software incorporates synchronization mechanisms with complex timing behaviors.
- Real-time processes execute at random times and preempt other processes at random points in time.
- Schedulers and other operating system processes such as interrupt-handling routines with complex behaviors affect application processes subtly and unpredictably.
- Programmers use ad hoc methods to deal with additional constraints on the applications such as precedence constraints, release times that are not equal to the beginning of their periods, and low-jitter requirements.
- Programmers use priorities to deal with every kind of requirement.
- To handle concurrent resource contention, programmers use task blocking, which might cause deadlocks.

Some of the most significant progress and most enduring results in software engineering were achieved through imposing restrictions on software structures.

When programmers combine these practices, the high complexity of the interactions between the different entities, and the sheer number of possible combinations of those interactions, significantly increase the chances that inspection and verification will overlook important cases.

Reducing complexity

The limitations I've just discussed tell us that, if the software and its timing behaviors are overly complex, determining whether the software satisfies the required timing properties might be practically impossible.

So how can we solve this problem? The most apparent answer would be to find ways to reduce software complexity.

Some of the most significant progress and most enduring results in software engineering were achieved through imposing restrictions on software structures. Examples include information hiding, abstract interfaces, hierarchical structuring and modular decomposition,[2–4] structured programming,[5] and organizing concurrent software as a set of cooperating sequential processes.[6]

The same general principle—imposing restrictions on software structures to reduce complexity—seems also to be the key to constructing software so that timing properties are easier to inspect and verify. This is the approach that preruntime scheduling uses.

Preruntime scheduling

Without loss of generality, suppose that the software we wish to inspect consists of a set of sequential programs. Some programs are to execute periodically, once in each time period. Some programs are to execute in response to asynchronous events.

Assume also that for each periodic program p, we know the

- Release time, r_p (the earliest time it can start its computation)
- Deadline, d_p (the time it must finish its computation)
- Worst-case computation time, c_p
- Period, prd_p

For each asynchronous program a, we know the

- Worst-case computation time, c_a
- Deadline, d_a
- Minimum time between two consecutive requests, $\min_a$

Furthermore, suppose some sections of some programs must precede a given set of sections in other programs. Also, some program sections might exclude a given set of sections of other programs. In addition, suppose that we know the computation time and start time of each program section relative to the beginning of the program containing that section. We assume that the worst-case computation time and each program's logical correctness have been independently verified.

The procedure

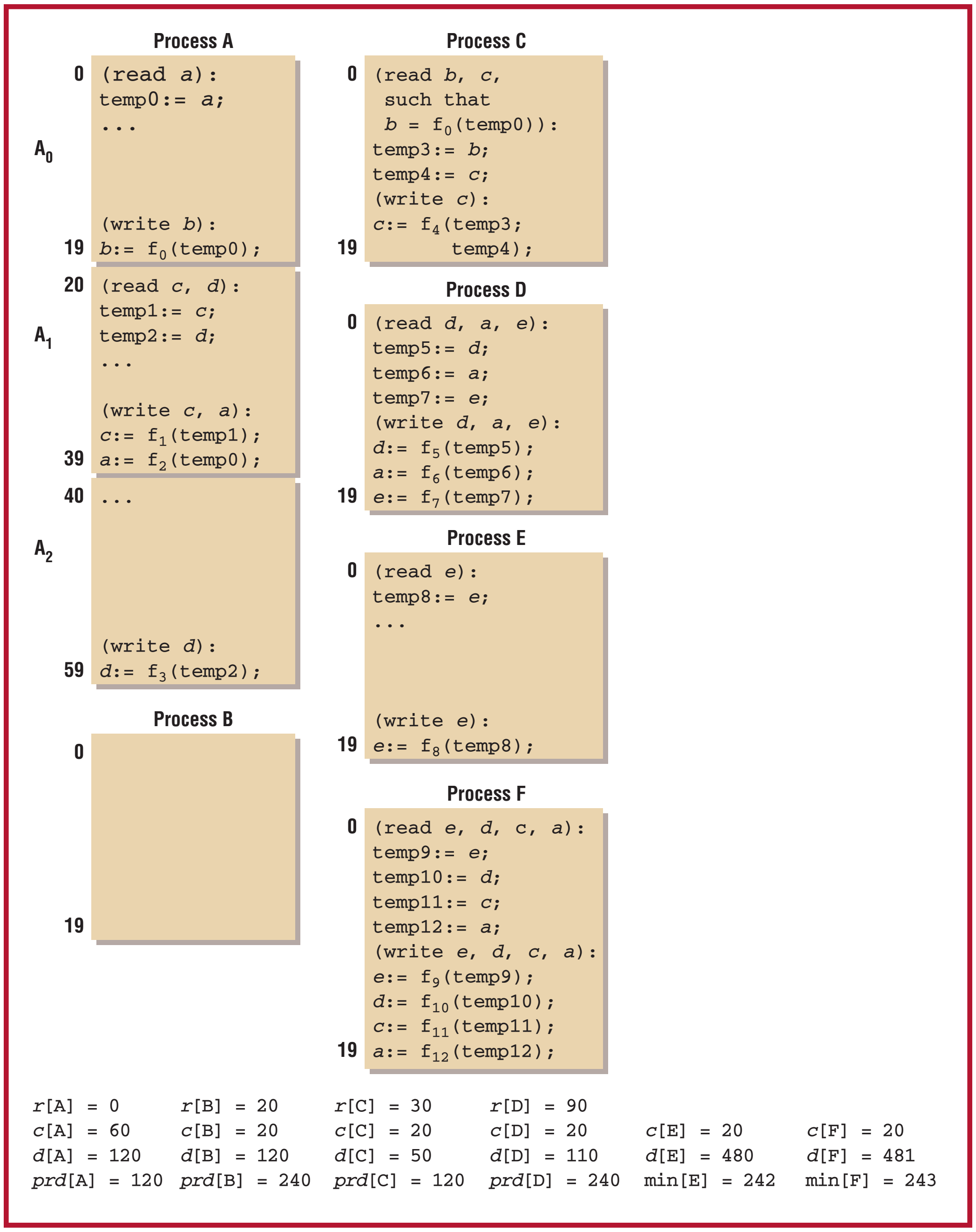
This approach comprises the following steps. First, organize the sequential programs as a set of cooperating sequential processes to be scheduled before runtime.

Second, identify all *critical sections*—that is, sections that access shared resources and sections that must execute before some sections of other programs, such as when a producer-consumer relation exists between sections. Divide each process into segments such that appropriate exclusion and precedence relations can be defined on pairs of sequences of the process segments to prevent simultaneous access to shared resources and to ensure proper execution order.

Third, convert each asynchronous process a into a new periodic process p. Suppose that P is the existing set of periods of periodic processes. One possible way to convert an asynchronous process a is to let the corresponding new periodic process p satisfy these conditions:

- $r_p = 0$
- $c_p = c_a$
- prd_p is equal to the largest member of P such that $2 \times prd_p - 1 \le d_a$ and $prd_p \le \min_a$
- d_p is equal to the largest integer such that $d_p + prd_p - 1 \le d_a$ and $d_p \le prd_p$
- $d_p \ge c_a$

Fourth, calculate each process segment's release time and deadline. For each process p with release time r_p, deadline d_p, and consisting of a sequence of process segments $p_0, p_1, \ldots, p_i, \ldots, p_n$, with computation times $c_{p_0}, c_{p_1}, \ldots, p_i, \ldots, c_{p_n}$, respectively, we can calculate the release time r_{p_i} and deadline d_{p_i} of each segment p_i as follows:



Process A

```
    0  (read a):
       temp0:= a;
       ...
A0
       (write b):
   19  b:= f0(temp0);
   20  (read c, d):
       temp1:= c;
A1     temp2:= d;
       ...

       (write c, a):
       c:= f1(temp1);
   39  a:= f2(temp0);
   40  ...

A2


       (write d):
   59  d:= f3(temp2);
```

Process B

```
    0


   19
```

Process C

```
    0  (read b, c,
        such that
        b = f0(temp0)):
       temp3:= b;
       temp4:= c;
       (write c):
       c:= f4(temp3;
   19         temp4);
```

Process D

```
    0  (read d, a, e):
       temp5:= d;
       temp6:= a;
       temp7:= e;
       (write d, a, e):
       d:= f5(temp5);
       a:= f6(temp6);
   19  e:= f7(temp7);
```

Process E

```
    0  (read e):
       temp8:= e;
       ...


       (write e):
   19  e:= f8(temp8);
```

Process F

```
    0  (read e, d, c, a):
       temp9:= e;
       temp10:= d;
       temp11:= c;
       temp12:= a;
       (write e, d, c, a):
       e:= f9(temp9);
       d:= f10(temp10);
       c:= f11(temp11);
   19  a:= f12(temp12);
```

```
r[A] = 0       r[B] = 20      r[C] = 30      r[D] = 90
c[A] = 60      c[B] = 20      c[C] = 20      c[D] = 20      c[E] = 20      c[F] = 20
d[A] = 120     d[B] = 120     d[C] = 50      d[D] = 110     d[E] = 480     d[F] = 481
prd[A] = 120   prd[B] = 240   prd[C] = 120   prd[D] = 240   min[E] = 242   min[F] = 243
```

Figure 1. The sequential programs A, B, C, and D are to execute periodically. The sequential programs E and F are to execute in response to asynchronous requests.

$$r_{p_i} = r_p + \sum_{j=0}^{i-1} c_{p_j}$$

$$d_{p_i} = d_p - \sum_{j=i+1}^{n} c_{p_j} .$$

Fifth, compute offline a preruntime schedule for all instances of the entire set of periodic segments occurring within a time period that is equal to the least common multiple of all periodic segments. This schedule should include new periodic segments converted from asynchronous segments. It should also satisfy all the release time, deadline, precedence, and exclusion relations.[7–10]

Sixth, at runtime execute all the periodic segments in accordance with the previously computed schedule.

Applying the approach

Suppose that in a hard-real-time system the software consists of six sequential programs A, B, C, D, E, and F, which are organized as a set of sequential processes that cooperate through reading and writing data on a set of shared variables *a*, *b*, *c*, *d*, and *e* (see Figure 1). A, B, C, and D are to execute periodically, with release times at 0, 20, 30, and 90 time units; computation times of 60, 20, 20, and 20 time units; deadlines at 120, 120, 50, and 110 time units; and periods of 120, 240, 120, and

$r[A_0] = 0$	$r[A_1] = 20$	$r[A_2] = 40$	$r[B] = 20$	$r[C] = 30$	$r[D] = 90$
$c[A_0] = 20$	$c[A_1] = 20$	$c[A_2] = 20$	$c[B] = 20$	$c[C] = 20$	$c[D] = 20$
$d[A_0] = 80$	$d[A_1] = 100$	$d[A_2] = 120$	$d[B] = 120$	$d[C] = 50$	$d[D] = 110$
$prd[A_0] = 120$	$prd[A_1] = 120$	$prd[A_2] = 120$	$prd[B] = 240$	$prd[C] = 120$	$prd[D] = 240$

$r[E'] = 0$	$r[F'] = 0$
$c[E'] = 20$	$c[F'] = 20$
$d[E'] = 240$	$d[F'] = 240$
$prd[E'] = 240$	$prd[F'] = 240$

(A_0,A_1) EXCLUDES (D)	(A_0,A_1) EXCLUDES (F)	(D) EXCLUDES (F)
(D) EXCLUDES (A_0,A_1)	(F) EXCLUDES (A_0,A_1)	(F) EXCLUDES (D)
(A_1,A_2) EXCLUDES (D)	(A_1,A_2) EXCLUDES (F)	(E) EXCLUDES (F)
(D) EXCLUDES (A_1,A_2)	(F) EXCLUDES (A_1,A_2)	(F) EXCLUDES (E)
(D) EXCLUDES (E)	(A_1) EXCLUDES (C)	(C) EXCLUDES (F)
(E) EXCLUDES (D)	(C) EXCLUDES (A_1)	(F) EXCLUDES (C)

A_0 PRECEDES C $\quad$ A_0 PRECEDES A_1 $\quad$ A_1 PRECEDES A_2

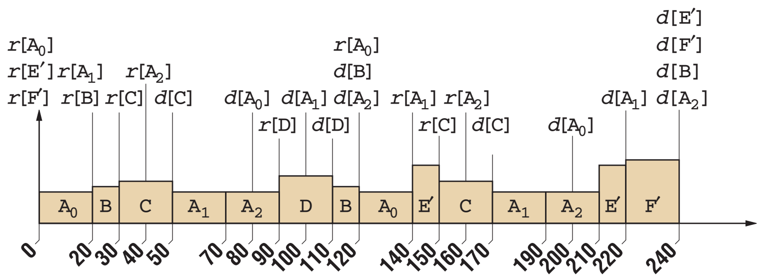


Figure 2. A preruntime schedule for the four periodic processes A, B, C, and D and the two asynchronous processes E and F.

240 time units, respectively. E and F are to execute in response to asynchronous requests, with computation times of 20 and 20 time units, deadlines at 480 and 481 time units, and a minimum time between two consecutive requests of 242 and 243 time units.

Process A, at the beginning of its computation, reads the current value of *a* and performs a computation based on that value. At this computation's 20th time unit, Process A writes a value into *b* that depends on the previously read value of *a*. This new value is intended to be read by Process C. At the 21st time unit, Process A also reads the current values of *c* and *d*. At the 40th time unit, Process A writes new values into *c* and *a* that depend on the previously read values of *c* and *a*. At the 60th time unit—the computation's last time unit—Process A writes a new value into *d* that depends on the previously read value of *d*.

To prevent processes from simultaneously accessing shared resources, such as the shared data in this example, we divide each process into a sequence of segments. We then define the critical sections, where each critical section consists of a sequence of the segments. We then define EXCLUDES (binary) relations between critical sections. For any satisfactory schedule, EXCLUDES relations must satisfy these conditions:

For any pair of critical sections x and y, if x EXCLUDES y, then no computation of any segment in y can occur between the time the first segment in x starts its computation and the time the last segment in x completes its computation.

To enforce the proper ordering of segments in a process, as well as producer-consumer relationships between segments belonging to different processes, we can define a PRECEDES relation on ordered pairs of segments. PRECEDES relations must satisfy these conditions:

For any pair of segments i and j, if i PRECEDES j, then j cannot start its computation before i has completed its computation.

(We expect that PRECEDES relations will usually be defined only on segments of processes in the same period.)

To prevent simultaneous access to *a*, *b*, *c*, *d*, and *e* and enforce the required producer-consumer relationship between A and C, while maximizing the chances of finding a feasible schedule, we can define the set of EXCLUDES and PRECEDES shown in Figure 2.

Using the conditions in the third step mentioned in the previous section, we can convert the asynchronous processes E and F into new periodic processes as follows:

```
r[E′] = 0, c[E′] = 20,
  d[E′] = 240, prd[E′] = 240.
r[F′] = 0, c[F′] = 20,
  d[F′] = 240, prd[F′] = 240.
```

Using the formula in the fourth step, we can calculate the release time and deadline of each segment in A. This calculation's results appear in Figure 2.

Given the EXCLUDES relation defined on overlapping critical sections and the PRECEDES relation defined on the process segments, an algorithm that can schedule processes with overlapping critical sections[9] should be able to find the feasible schedule shown in Figure 2. In this schedule, all instances of all the process segments of A_0, A_1, A_2, C, B, D, E′, and F′ occurring within the preruntime schedule length of 240 time units meet their deadlines. Also, this schedule satisfies all the specified EXCLUDES and PRECEDES relations.

Why preruntime scheduling works

Here's why this approach makes software timing properties easier to inspect and verify.

First, instead of having to exhaustively analyze and inspect a huge number of different possible interleaving or concurrent task-execution sequences, you need to inspect only a single preruntime schedule each time.

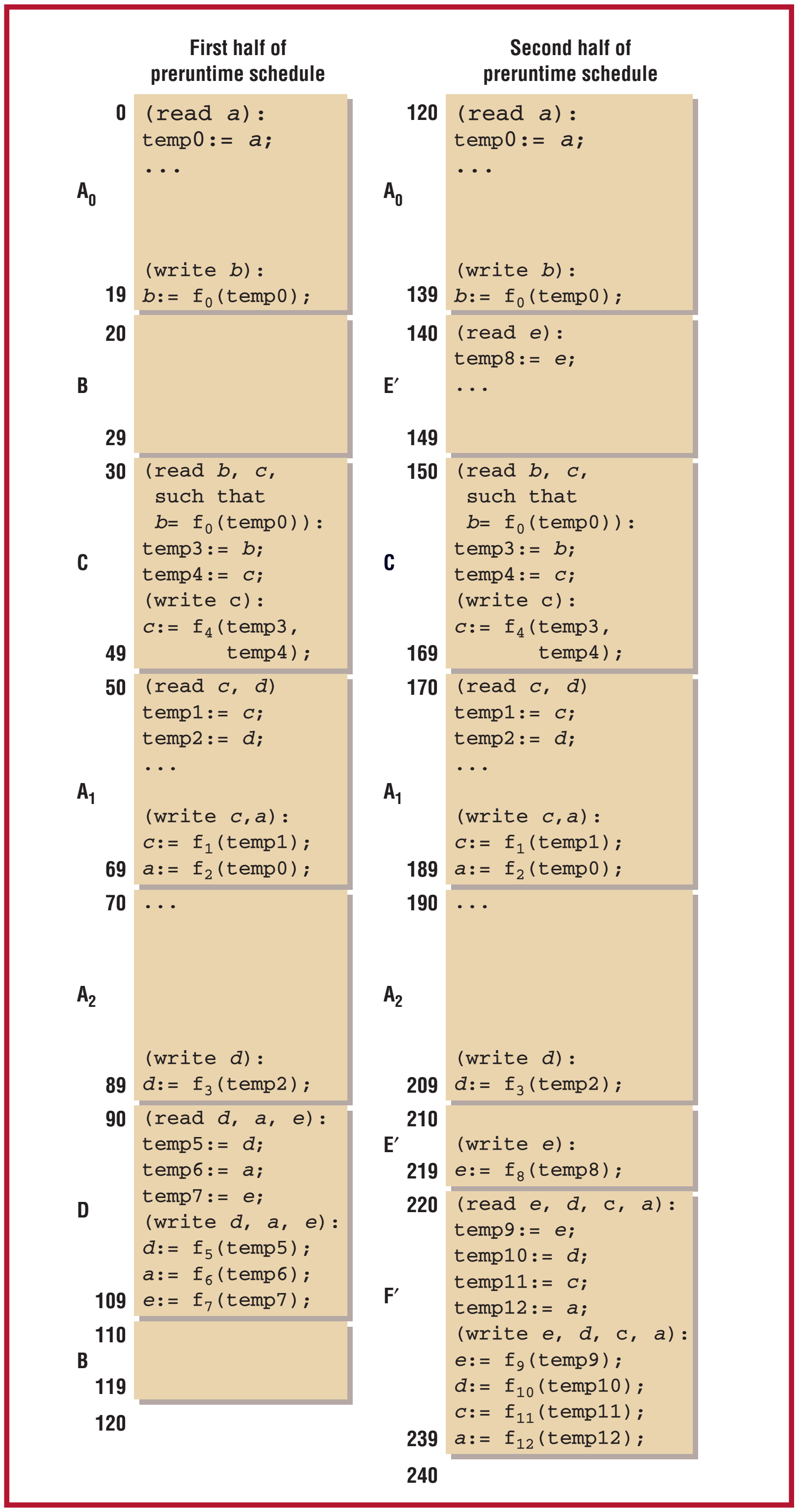
Second, in each preruntime schedule, the interleaving or concurrent task-execution sequence is statically and visually laid out in one straight line of actual code (see Figure 3). So, you can easily verify, by straightforward visual inspection of the schedule, that the execution sequence meets all the timing constraints such as release times and deadlines, periods, and low-jitter requirements.

Third, instead of using complex, unpredictable runtime synchronization mechanisms to prevent simultaneous access to shared data, the preruntime scheduling approach simply constructs preruntime schedules in which critical sections that exclude each other do not overlap. This lets you easily verify visually that the execution sequence meets requirements such as exclusion relations and precedence relations between actual code segments of real-time tasks.

Fourth, instead of having to assume that context switches can happen at any time, you can easily verify visually exactly when, where, and how many context switches might happen.

Fifth, task deadlocks cannot occur.

Sixth, you can switch processor execution from one process to another through simple mechanisms such as procedure calls or simply by catenating code, which reduces system overhead and simplifies the timing analysis.



First half of preruntime schedule
Second half of preruntime schedule

```
First half of preruntime schedule

      0   (read a):
          temp0:= a;
          ...
A0
          (write b):
     19   b:= f0(temp0);
     20
B
     29
     30   (read b, c,
           such that
           b= f0(temp0)):
          temp3:= b;
C         temp4:= c;
          (write c):
          c:= f4(temp3,
     49           temp4);
     50   (read c, d)
          temp1:= c;
          temp2:= d;
          ...
A1
          (write c,a):
          c:= f1(temp1);
     69   a:= f2(temp0);
     70   ...
A2
          (write d):
     89   d:= f3(temp2);
     90   (read d, a, e):
          temp5:= d;
          temp6:= a;
D         temp7:= e;
          (write d, a, e):
          d:= f5(temp5);
          a:= f6(temp6);
    109   e:= f7(temp7);
    110
B
    119
    120
```

```
Second half of preruntime schedule

    120   (read a):
          temp0:= a;
          ...
A0
          (write b):
    139   b:= f0(temp0);
    140   (read e):
          temp8:= e;
E′        ...
    149
    150   (read b, c,
           such that
           b= f0(temp0)):
          temp3:= b;
C         temp4:= c;
          (write c):
          c:= f4(temp3,
    169           temp4);
    170   (read c, d)
          temp1:= c;
          temp2:= d;
          ...
A1
          (write c,a):
          c:= f1(temp1);
    189   a:= f2(temp0);
    190   ...
A2
          (write d):
    209   d:= f3(temp2);
    210
E′        (write e):
    219   e:= f8(temp8);
    220   (read e, d, c, a):
          temp9:= e;
          temp10:= d;
          temp11:= c;
F′        temp12:= a;
          (write e, d, c, a):
          e:= f9(temp9);
          d:= f10(temp10);
          c:= f11(temp11);
    239   a:= f12(temp12);
    240
```

Figure 3. The detailed code layout in the preruntime schedule for the periodic processes A, B, C, and D and the asynchronous processes E and F makes inspecting the programs' timing and runtime behavior much easier.

Some people think that preruntime scheduling is less flexible than the alternatives. I believe these perceptions are mostly misconceptions.

Seventh, an automated preruntime scheduler can help automate and speed up important parts of inspection. Whenever you need to modify a program, a new preruntime schedule can be automatically and quickly generated, letting you quickly learn whether the modifications affect any timing requirements.

Finally, you can convert most asynchronous processes to periodic processes. So, you can use periodic processes instead of interrupts to handle external and internal events. This removes a significant source of nondeterministic behavior from the system, which greatly reduces the complexity of the software and its timing behaviors. Terry Shepard and Martin Gagne have applied this preruntime-scheduling approach to the F-18 Aircraft Mission Computer on Flight Program.[11]

Misconceptions about preruntime scheduling

Some people think that preruntime scheduling is less flexible than the alternatives. I believe these perceptions are mostly misconceptions.

First, some people think that once you have scheduled process segments into a preruntime schedule, modifying the system to meet new requirements will be difficult. This perception originated with the earlier, more primitive form of preruntime schedules—*cyclic executives*. System designers had to construct them completely by hand because suitable algorithms to automate the task were not available. Also, because of the difficulty of rescheduling processes to obtain a new cyclic executive, whenever system changes were required, the designers would directly modify the existing cyclic executive. After a few modifications, the original processes' logical structure would be lost. This loss made the system difficult to understand and further modify, resulting in a system that designers described as "fragile."

Preruntime scheduling algorithms[7–10] provide a different approach to building real-time systems; they can completely automate construction of preruntime schedules. This lets the designer always maintain the system structures in two distinct but corresponding levels:

- *The higher logical level* consists of the original cooperating sequential processes and the various logical constraints, including timing constraints defined on those processes.
- *The lower implementation level* consists of the preruntime schedule—that is, the execution ordering of those processes.

Whenever system changes are required, the designer doesn't directly alter the preruntime schedule at the lower implementation level. Instead, he or she modifies the original cooperating sequential processes at the higher logical level, using the higher-level knowledge about the logical structures of the processes and the logical constraints on them. After the modifications are complete, the designer can use the preruntime scheduling algorithms to automatically reschedule the modified processes and segments, to obtain a new preruntime schedule. This lets designers keep intact any desired logical properties in the original process structures that are useful for understanding, maintaining, and reasoning about the programs' properties and correctness. They can also use those logical properties to make further modifications or add new features or processes to the system at the higher logical level.

Second, some people have claimed that preruntime scheduling is not as flexible as fixed-priority-based schedulers because it does not allow dynamic admission of tasks. Actually, no runtime scheduler will ever be able to guarantee that the timing constraints of an arbitrary set of dynamically arriving tasks can be satisfied, without knowing in advance information about such tasks' characteristics.[12] However, if you know this information in advance, there's no reason why you shouldn't use it to the maximum extent to determine before runtime whether those timing constraints can be satisfied, even if this requires substantial offline computation.

But runtime fixed-priority-based synchronization protocols and mechanisms, such as rate-monotonic scheduling and the Priority Ceiling Protocol, can't take into account much of the information that is known before runtime. They can't handle complex application constraints, such as release times, precedence constraints, and low-jitter requirements. They generally result in lower processor utilization, have much greater system overhead, and make the system's runtime behavior significantly more difficult to analyze and predict. Detailed discussions of these issues appear elsewhere.[13,14]

Also, a preruntime schedule dispatcher such as the one I've described in a previous paper[14] can use the time that is unused by time-critical tasks to execute non-time-critical dynamically arriving tasks.

To guarantee that a large-scale, complex, safety-critical real-time system will not fail unexpectedly, we must be able to predict all the possible cases of the actual time-critical software code's timing behaviors through rigorous inspection and verification.

However, in most cases, researchers verify the timing properties of only specifications, models, algorithms, and protocols, not of the actual code. In most cases, they give no proof that the specifications, models, algorithms, and protocols have the same timing properties as the actual code. When researchers attempt to "verify" timing properties of actual code in existing real-time systems where processes are scheduled at runtime, typically they

- Can't handle more than a small number of processes
- Can't simultaneously handle common real-time application constraints and interdependencies such as release times, precedence, and exclusion relations between the processes
- Can't take into account all the implementation details of the system that affect timing

But even with these severe limitations, they still can't cover more than an extremely small subset of the possible cases of the actual code's timing behaviors.

The overwhelmingly large number of possible cases of existing real-time software's timing behaviors is due largely to overly complex interactions between system components. The preruntime scheduling approach effectively reduces the number of the possible cases of the actual code's timing behaviors by structuring real-time software as a set of cooperating sequential processes and imposing strong restrictions on the interactions between the processes. This makes it easier to inspect and verify all the timing behaviors of the software.

I still need to ensure that each sequential process's worst-case execution time can be independently inspected and verified. I believe that a key to ensuring this is to also impose strong restrictions on the sequential program structures. I plan to discuss these issues in a future paper.

About the Author

Jia Xu is an associate professor of computer science at York University in Toronto. His current research interest is real-time and embedded-systems engineering. He received his Docteur en Sciences Appliquées in computer science from the Université Catholique de Louvain. Contact him at the Dept. of Computer Science, York Univ., 4700 Keele St., North York, ON M3J 1P3, Canada.

Acknowledgments

A Natural Sciences and Engineering Council of Canada grant partially supported this work. I presented a preliminary version of this article at the First Workshop on Inspection in Software Engineering (WISE 01). I thank the reviewers for numerous thoughtful comments and helpful suggestions on how to improve this article. I am also indebted to Dave Parnas for helpful discussions and advice related to this work.

References

1. R. Alur and T.A. Henzinger, "Logics and Models of Real Time: A Survey," *Real Time: Theory in Practice*, J.W. de Bakker et al., eds., LNCS 600, Springer-Verlag, 1992, pp. 74–106.
2. D.L. Parnas, "On the Criteria Used in Decomposing Systems into Modules," *Comm. ACM*, vol. 15, no. 12, Dec. 1972, pp. 1053–1058.
3. D.L. Parnas, "On a 'Buzzword': Hierarchical Structure," *Proc. IFIP Congress 74*, North Holland Publishing, 1974, pp. 336–339.
4. D.M. Hoffman and D.M. Weiss, eds., *Software Fundamentals: Collected Papers by David L. Parnas*, Addison-Wesley, 2001.
5. E.W. Dijkstra, "Structured Programming," *Software Engineering Techniques*, J.N. Buxton and B. Randell, eds., NATO Scientific Affairs Div., 1970, pp. 84–87.
6. E.W. Dijkstra, "Cooperating Sequential Processes," *Programming Languages,* F. Genuys, ed., Academic Press, 1968, pp. 43–112.
7. J. Xu and D.L. Parnas, "On Satisfying Timing Constraints in Hard-Real-Time Systems," *IEEE Trans. Software Eng.*, vol. 19, no. 1, Jan. 1993, pp. 70–84. Reprinted in *A Practical Approach to Real-Time Systems: Selected Readings*, Philip Laplant, ed., IEEE Press, 2000, pp. 15–31.
8. J. Xu, "Multiprocessor Scheduling of Processes with Release Times, Deadlines, Precedence, and Exclusion Relations," *IEEE Trans. Software Eng.*, vol. 19, no. 2, Feb. 1993, pp. 139–154.
9. J. Xu and D.L. Parnas, "Pre-Run-Time Scheduling of Processes with Exclusion Relations on Nested or Overlapping Critical Sections," *Proc. 11th Ann. IEEE Int'l Phoenix Conf. Computers and Communications* (IPCCC 92), IEEE Press, 1992, pp. 774–782.
10. J. Xu and D.L. Parnas, "Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations," *IEEE Trans. Software Eng.*, vol. 16, no. 3, Mar. 1990, pp. 360–369. Reprinted in *Advances in Real-Time Systems*, J.A. Stankovic and K. Ramamrithan, eds., IEEE CS Press, 1993, pp. 140–149.
11. T. Shepard and M. Gagne, "A Model of the F-18 Mission Computer Software for Pre-Run-Time Scheduling," *Proc. 10th Int. Conf. Distributed Computing Systems* (ICDCS 90), IEEE CS Press, 1990, pp. 62–69.
12. A.K. Mok, "The Design of Real-Time Programming Systems Based on Process Models," *Proc. 1984 IEEE Real-Time Systems Symp.*, IEEE CS Press, 1984, pp. 5–17.
13. J. Xu and D.L. Parnas, "Priority Scheduling versus Pre-Run-Time Scheduling," *Real-Time Systems*, vol. 18, no. 1, Jan. 2000, pp. 7–23.
14. J. Xu, "On Inspection and Verification of Software with Timing Requirements," to be published in *IEEE Trans. Software Eng.*, vol. 29, no. 8, Aug. 2003.