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]

0740-7459/03/$17.00 © 2003 IEEE

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.

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.

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.

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 $2 \times prd_p - 1 \leq d_a$ and $prd_p \leq min_a$
- d_p is equal to the largest integer such that $d_p + prd_p - 1 \leq d_a$ and $d_p \leq prd_p$
- $d_p \geq c_a$

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 $c_{p_0}, c_{p_1}, \ldots, c_{p_i}, \ldots, c_{p_n}$, respectively, we can calculate the release time r_{p_i} and deadline d_{p_i} of each segment p_i as follows: