On Inspection and Verification of Software With Timing Requirements

Jia Xu

Abstract— **Many current practices in writing real-time software make it extremely hard, if not impossible, to verify that the resulting software satisfies given timing requirements. However, if certain restrictions are imposed on the software structure, inspection of the software for timing is much easier. We discuss procedures applying the restrictions that greatly simplify the task of inspecting software with timing requirements.**

Keywords— **Software inspection, verification, timing requirements, current practices, complexity, restrictions, software structure, pre-run-time scheduling.**

I. INTRODUCTION

MORE and more infrastructure of the world is becoming dependent on computer systems that have timing requirements. Communications, transportation, medicine, energy, finance, defense, are all increasingly involving processes and activities that require increasingly precise observance of timing constraints. One likely reason for this unrelenting trend towards real-time synchronization and coordination, is that tighter synchronization and coordination of processes and activities, generally result in higher efficiency, throughput and productivity. A global reference is needed, if a large number of different processes and activities are to be coordinated and synchronized on a global basis. Time provides such a global reference. Real-time software is often required to handle the coordination and synchronization of many different processes and activities. This seem to be one of the reasons why real-time and embedded software that must observe timing constraints are experiencing explosive growth.

In contrast, there is a conspicious lack of effective methods and tools for verifying timing properties of software, despite an increasingly pressing need for such methods and tools.

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

Timing requirements and constraints in concurrent, real-time software pose special problems for software inspection and verification. The basic premise of any software inspection or verification method, is that it should be able to cover all the possible cases of the software's behaviour. Taking into account timing parameters and constraints adds a whole new dimension to the solution space. The number of different possible interleaving and/or concurrent execution sequences of multiple threads of software processes and activities that need to be considered when timing constraints are included, may increase exponentially as a function of the size of the software, and may result in an explosion of the number of different cases that needs to be examined. This may make it exceedingly difficult to use verification and inspection techniques that systematically examine all the possible cases of program behaviour.

Examples of proposed formal methods for real-time systems, include, amongst others, timed automata [2], timed transition systems/temporal logic [7] [13] [8], Modecharts [9], theorem proving techniques using PVS to analyse real-time scheduling protocols [6]. The common difficulty of applying these methods to actual real-time software (as opposed to simplified high level abstractions of algorithms/protocols which are only approximations of the actual software and which do not take into account all the implementation details that may affect timing) is the exponential blowup in complexity. As discussed in [2], in any precise model that attempts to capture all the possible cases of software behaviour, the complexity is proportional to the number of states in the global timed structure describing the software implementation. If no restriction is imposed on the software structure, then the number of states is exponential in the number of system components, making it impractical to apply these methods to cover all the possible states corresponding to large scale complex real-time software's overwhelmingly large number of possible timing behaviours.

II. IMPOSING RESTRICTIONS ON SOFTWARE STRUCTURE TO REDUCE COMPLEXITY

If the problem is the complexity of software and its overwhelmingly large number of possible timing behaviours, then what can be done? The most apparent answer would be to find ways to reduce that complexity.

Perhaps a parallel can be drawn between the problem of constructing software that is in general easier to understand, prove correct, and maintain; and the problem of inspecting and verifying the timing of real-time software.

Some of the most significant progress and most enduring results in software engineering were achieved through imposing restrictions on software structure. Information hiding, the idea of abstract interfaces, the idea of hierarchical structuring and modular decomposition [13], [14]; the ideas of structured programming [5], and organizing concurrent software as a set of cooperating sequential processes [4], are examples.

Current address: Department of Computer Science, York University, 4700 Keele Street, North York, Ontario M3J 1P3, Canada.

This work was partially supported by a Natural Sciences and Engineering Council of Canada grant.

People came to understand the importance of imposing restrictions on software structure.

- Without appropriate structure, the interactions between code sections are unrestricted. Any part of the program could cause a serious failure.

- Without appropriate structure, the software becomes a sea of details, and it becomes extremely difficult to be convinced that that sea of details fits together to achieve the required functionality.

- Without appropriate structure, it is far less likely that an inspection method would be effective in verifying a complex software's properties through "divide and conquer" [15]. Spaghetti code will make it difficult to decompose the code in a way such that each partition of the code could be studied independently of the rest of the code.

- Without putting restrictions on the ways in which concurrent processes interact and communicate, race conditions occur in which the outcome of the execution depends on the particular order in which access to shared data takes place.

The same general principle, imposing restrictions on software structure to reduce complexity, seems also to be the key to constructing software so that timing properties can be easily inspected and verified.

Software with hard timing requirements should be designed using a systematic approach to make their timing properties verifiable and easy to inspect. There will probably never exist a general method or tool that can verify the timing properties of any arbitrary piece of software, just like it is unlikely that general methods or tools will prove effective in verifying properties of a badly structured program consisting of "spaghetti" code woven together with "goto" statements.

III. CURRENT PRACTICES IN THE DESIGN OF REAL-TIME SOFTWARE THAT MAKE IT DIFFICULT TO INSPECT AND VERIFY TIMING PROPERTIES

In the following, we mention current practices in the design of real-time nonterminating and concurrent software that make it more difficult to verify and inspect timing properties, very much like the unrestricted use of "goto" statements destroy the structure of regular programs and make it more difficult to inspect and verify their properties.

(a) Complex synchronization mechanisms are used in order to prevent simultaneous access to shared resources. These synchronization mechanisms often use queueing constructs where queueing policies such as FIFO and blocking can make the timing behaviour unpredictable.

(b) Real-time processes not only execute at random times; they are often allowed to preempt other processes at random points in time. Not only the context switch times vary, but it also results in a huge increase in the number of different possible execution interleaving sequences, many of which may have unpredictable effects on timing.

(c) The execution of run-time schedulers and other operating system processes such as interrupt handling routines with complex behaviours (and often with the highest priorities) interleave with the execution of real-time application processes, affecting the timing of the application processes in subtle and unpredictable ways.

(d) When many additional constraints are required by the application, such as precedence constraints, release times that are not equal to the beginning of their periods, low jitter requirements, etc., are added to the timing constraints, because current run-time scheduling algorithms and mechanisms are unable to solve such problems, practitioners use ad hoc run-time methods to try to satisfy the additional constraints. These ad hoc run-time methods tend to affect timing in highly unpredictable ways.

(e) Priorities are used to try to deal with every kind of requirement [17]. The priority assignments often conflict with other application requirements. In practice, task priorities are rarely application requirements, but are used instead as the primary means for trying to meet timing constraints. These priorities frequently change, which greatly complicates the timing analysis.

(f) Task blocking is used to handle concurrent resource contention, which, in addition to making the timing unpredictable, may result in deadlocks.

Even in fairly simple systems in which a few of the above practices are used, inspecting software with timing constraints can still be a very daunting task. For example, in one study fixed priority scheduling was implemented using priority queues, where tasks were moved between queues by a scheduler that was ran at regular intervals by a timer interrupt. It had been observed that, because the clock interrupt handler had a priority greater than any application task, even a high priority task could suffer long delays while lower priority tasks were moved from one queue to another. Accurately predicting the scheduler overhead proved to be an extraordinarily complicated task, even though the system was very simple, with a total of only 20 tasks, where tasks did not have critical sections, priorities do not change, and the authors of the study are considered to be among the world's foremost authorities on priority scheduling [3].

When the above current practices are used in combination with each other, the high complexity of the interactions between the different entities, and the sheer number of different possible combinations of those interactions, significantly increase the chances that some important cases will be overlooked in the inspection process.

If some of the world's top experts on priority scheduling have such difficulty predicting the timing behaviour of such a small and limited system, it would indeed be very difficult for most people to be able to inspect and accurately verify the timing behaviour of large scale, complex, nonterminating, and concurrent software written using the current practices just described.

IV. RESTRICTIONS ON SOFTWARE STRUCTURE THAT WILL SIMPLIFY INSPECTION AND VERIFICATION OF TIMING

We have observed from experience that the following set of restrictions on software structure will simplify inspection and verification for timing.

(1) The software is structured as a set of cooperating sequential processes.

(2) Each process is divided into a sequence of segments, according to exclusion relations[1] and precedence relations[2] defined on the process segments.

(3) The number ($N_{ordering}$) of different possible relative orderings of the run-time process segment executions in each Least Common Multiple of the periods of the set of periodic processes (including asynchronous processes that are translated to periodic processes) in each system mode is a relatively small constant in relation to the number of process segments. That is, the value $N_{ordering}$ should largely be independent of the number of process segments; the value $N_{ordering}$ should not increase greatly when the number of process segments grows[3].

If the above restrictions on software structure are satisfied, then the inspection and verification of timing will be simplified because:

(a) The number of different cases of timing of the software that needs to be inspected to verify that all timing constraints will be satisfied is $O(N_{ordering})$ in the worst case.

(b) The number of different cases of timing of the software that needs to be inspected to verify that all exclusion relations and precedence relations will be satisfied is $O(N_{ordering})$ in the worst case.

(c) The number of different cases of timing of the software that needs to be inspected to verify that deadlock will not happen is $O(N_{ordering})$ in the worst case.

It should not be hard to see that each of the current practices mentioned in the previous section do not satisfy the restrictions above.

Applying the restrictions should significantly reduce the number of different states in the global timed structure describing the software implementation, and should help in preventing the kind of blowup in complexity which in the past has been one of the main obstacles to applying formal methods to model and verify timing properties of software mentioned near the end of the first section of this paper.

[1]A process segment i is said to *exclude* another process segment j if no execution of j can occur between the time that i starts its computation and the time that i completes its computation. Exclusion relations may exist between process segments when some process segments must prevent simultaneous access to shared resources such as data and I/O devices by other process segments.

[2]A process segment i is said to *precede* another process segment j if j can only start execution after i has completed its computation. Precedence relations may exist between process segments when some process segments require information that is produced by other process segments.

[3]This also implies that $N_{ordering}$ should not increase greatly when aperiodic processes are translated into periodic processes.

V. A PROCEDURE FOR STRUCTURING REAL-TIME SOFTWARE THAT WILL SIMPLIFY INSPECTION AND VERIFICATION OF TIMING

Below we describe a procedure, which we call *pre-run-time scheduling*, for structuring real-time software, which will simplify inspection and verification of timing, because the procedure satisfies the restrictions listed in Section 4.

Without loss of generality, suppose that the software we wish to inspect for timing consists of a set of sequential programs. Some of the programs are to be executed periodically, once in each period of time. Some of the programs are to be executed in response to asynchronous events. Assume also that for each periodic program we are given the earliest time that it can start its computation, called its release time; the deadline by which it must finish its computation; its worst-case computation time; and its period. For each asynchronous program we are given its worst-case computation time, its deadline, and the minimum time between two consecutive requests. Furthermore, suppose there may exist some sections of some programs that are required to precede a given set of sections in other programs. This may happen when there is a producer-consumer relationship between the two sections. There also may exist some sections of some programs that exclude a given set of sections of other programs, i.e., once a section has started its computation it cannot be preempted by any section in the set that it excludes. This may happen when the two sections read and write common data. Also suppose that we know the computation time and start time of each program section relative to the beginning of the program containing that section.

The procedure consists of the following steps:

(1) Organize the sequential programs as a set of sequential processes to be scheduled before run-time.

(2) Divide each process into process segments according to the precedence and exclusion relations defined on the pairs of program sections.

(3) Calculate the release time and deadline for each segment.

(4) Translate asynchronous segments into periodic segments using the algorithm in [11] [12].

(5) Compute off-line a schedule, called a *pre-run-time schedule* for the entire set of periodic segments, including new periodic segments translated from asynchronous segments, occurring within a time period that is equal to the

least common multiple[4] of all periodic segments, which satisfies all the release time, deadline, precedence, and exclusion relations [21] [20] [18] [19].

(6) At run-time schedule all the periodic segments in accordance with the previously computed schedule.

By following this procedure, all the restrictions in the previous section are easily satisfied. This makes it very easy to inspect the timing behaviour of all the processes, and verify that all the timing requirements will be satisfied. (See Fig. 1 for an example of a pre-run-time schedule)

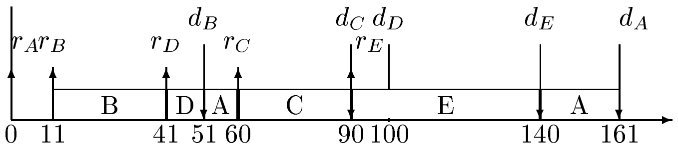

Fig. 1. A pre-run-time schedule for the 5 process segments A, B, C, D, E, that satisfies the timing constraints: $r_A = 0, c_A = 30, d_A = 161; r_B = 11, c_B = 30, d_B = 51$; $r_C = 60, c_C = 10, d_C = 90; r_D = 41, c_D = 10$, $d_D = 100; r_E = 90, c_E = 50, d_E = 140$; and satisfies the precedence relations: B PRECEDES D; and satisfies the exclusion relations: A EXCLUDES D, A EXCLUDES B, B EXCLUDES C, C EXCLUDES E, C EXCLUDES D, D EXCLUDES E.

(a) Instead of having to exhaustively analyze and inspect a huge number of different possible interleaving/concurrent task execution sequences, with a pre-run-time scheduling approach, one only needs to inspect one single pre-run-time schedule each time.

(b) In each pre-run-time schedule, the interleaving/concurrent task execution sequence is statically and visually laid out in one straight line of code. This makes it easy to verify, by straightforward visual inspection of the pre-run-time schedule, that all the timing constraints such as release times and deadlines, periods, low jitter requirements, etc., are met by the execution sequence.

(c) Instead of using complex, unpredictable run-time synchronization mechanisms to prevent simultaneous access to shared data, the pre-run-time scheduling approach prevents simultaneous access to shared data simply by constructing pre-run-time schedules in which critical sections that exclusion each other do not overlap in the schedule. This makes it easy to verify, by straightforward visual inspection of the pre-run-time schedule, that requirements such as exclusion relations and precedence relations between code segments of real-time tasks, are met by the execution sequence.

(d) Instead of having to assume that context switches can happen at any time, it is easy to verify, by straightforward visual inspection of the pre-run-time schedule, exactly when, where and how many context switches may happen.

[4]When the process periods are relatively prime, the Least Common Multiple (LCM) of the process periods and the length of the pre-run-time schedule may become inconveniently long. However, in practice, one often has the flexibility to adjust the period lengths in order to obtain a satisfactory length of the LCM of the process periods. While this may result in some reduction in the processor utilization, the reduction should be insignificant when compared to the decrease in processor utilization with priority scheduling.

(e) In a pre-run-time schedule, there is no possibility of task deadlocks occurring.

(f) With a pre-run-time scheduling approach, one can switch processor execution from one process to another through very simple mechanisms such as procedure calls, or simply by catenating code when no context needs to be saved or restored, which simplifies the timing analysis.

(g) With a pre-run-time scheduling approach, an automated pre-run-time scheduler can help automate and speed up important parts of the inspection process. Whenever a program needs to be modified, a new pre-run-time schedule can be automatically and quickly generated, allowing one to quickly learn whether any timing requirements are affected by the modifications.

(h) With a pre-run-time scheduling approach, a "divide-and-conquer" approach can be applied. By a simple visual inspection of the pre-run-time schedule to identify which group of tasks form a continuous utilization of a processor, one can easily draw conclusions regarding which tasks' timing characteristics affect which other tasks' timing characteristics. For example, if a task misses its deadline in the pre-run-time schedule, the schedule allows one to immediately focus attention on those tasks whose timing characteristics, if changed, may allow that task to meet its deadline, while ignoring those tasks that will have no impact on the task in question.

(i) Pre-run-time scheduling can make it easier to implement a relatively constant "loop time" for control systems software, making it easier to implement and verify timing properties that span longer durations by making use of the loop time; and it can also help to reduce jitter in the output and guarantee constant sampling times for inputs both of which can be critical to guarantee stability of feedback control systems.

(j) Because in each pre-run-time schedule, the interleaving/concurrent task execution sequence is statically laid out in one straight line of code, the number of different states in the global timed structure describing the software implementation should be significantly reduced, and this should simplify the use of formal methods to verify timing properties.

The pre-run-time scheduling approach has numerous other important advantages, including:
(i) ability to effectively handle complex constraints and dependencies;
(ii) lower run-time overhead;
(iii) higher processor utilization, etc.

Since we are mainly concerned with inspecting and verifying the software's timing behaviour, readers are directed to [17] for a more detailed discussion on the other advantages of the pre-run-time scheduling approach not discussed in this paper. For another example of a procedure using the pre-run-time scheduling approach that can handle many different types of processes, while satisfying the restrictions listed in Section 4, see [16].

In addition to pre-run-time scheduling, there exist other important aspects of good modular design (e.g. clean interfaces to parts of the system that implement timers) that can simplify both inspection and verification, which we plan to discuss in a future version of this paper.

VI. SUMMARY

Many current practices in writing real-time software make it extremely hard, if not impossible, to verify that the resulting software satisfies given timing requirements. However, if certain restrictions are imposed on the software structure, inspection of the software for timing is much easier. We discuss procedures applying the restrictions that greatly simplify the task of inspecting software with timing requirements.

VII. APPENDIX: DISCUSSION ON PERCEIVED DISADVANTAGES OF THE PROPOSED METHOD.

Some people may have the perception that pre-run-time scheduling is less flexible, requires more effort to design, and requires more execution time, compared with alternatives, mainly priority scheduling schemes. In the following we shall provide reasons for which we believe that these perceptions are mostly misconceptions.

1. There is a perception that once the process segments have been scheduled into a pre-run-time schedule, then it will be difficult to modify the system to meet new requirements, making the system inflexible and requiring more design effort. This perception was actually created in the past when the earlier and more primitive form of pre-run-time schedules - cyclic executives, were constructed completely by hand because of a lack of suitable algorithms to automate the task. Once the cyclic executive schedule was generated, because of the difficulty of the task of rescheduling the processes to obtain a new cyclic executive schedule, whenever changes to the system were required, system designers would directly modify the cyclic executive schedule, causing the original processes' logical structure to be lost in the schedule after a few modifications, making it very difficult to understand and very difficult to further modify the system, resulting in a system described as "fragile" by designers.

With the availability of more recently developed pre-run-time scheduling algorithms [21] [20] [18] [19], real-time systems can be built using a different approach. The task of constructing pre-run-time schedules can now be completely automated using the pre-run-time scheduling algorithms. This allows the designer to always maintain the system structure in two distinct but corresponding levels - a *higher logical level* consisting of the original cooperating sequential processes and the various logical constraints including timing constraints defined on those processes; and a *lower implementation level* consisting of the pre-run-time schedule, that is, the execution ordering of those processes. Whenever changes to the system are required, instead of directly altering the pre-run-time schedule at the lower implementation level, the designer modifies the original cooperating sequential processes at the higher logical level, using the higher level knowledge about the logical structure of the processes and the logical constraints on them. After the modifications on the processes have been completed, the designer can use the pre-run-time scheduling algorithms to automatically reschedule the modified processes and segments, to obtain a new pre-run-time schedule. This allows designers to always keep intact any desired logical properties in the original process structures that are useful for understanding, maintaining and reasoning about the properties and correctness of the programs, and use those logical properties to continue to make further modifications or add new features/processes to the system at the higher logical level.

2. One of the reasons for which we believe that the pre-run-time scheduling approach actually provides more flexibility than the priority scheduling approach, is that with the priority scheduling approach, the execution orderings of processes are constrained by the rigid hierarchy of priorities that are imposed on processes, whereas with the pre-run-time scheduling approach, there is no such constraint — the system designer can switch from any pre-run-time schedule to any other pre-run-time schedule in any stage of the software's development. Here are a few examples.

(a) It has been frequently claimed that the priority scheduling approach has superior "stability" compared with other approaches, because "essential" processes can be assigned high priorities in order to ensure that they meet their deadlines in transient system overload situations [10]. The Rate Monotonic Scheduling approach assigns higher priorities to processes with shorter periods, because it has been proved that, if processes with longer periods are assigned higher priorities, then the schedulability of the whole system will be severely reduced. However, essential processes may not have short periods. While suggestions like cutting essential processes into smaller processes that are treated as processes with short periods have been made[5], these suggestions not only increase run-time overhead, but also add new artificial constraints to the problem, which increase the complexity and reduce the schedulability of the whole system. In real-time applications, under different circumstances, different sequences of process execution may be required, and sometimes different sets of processes become "essential." This is a problem which cannot easily be solved by assigning a rigid hierarchy of priorities to processes.

A pre-run-time scheduling approach can guarantee essential processes just as well, or better, than a priority scheduling approach. When using the pre-run-time scheduling approach, in the case of system overload, an alternative pre-run-time schedule which only includes the set of processes that are considered to be essential under the particular circumstances can be executed. As pre-run-time schedules

[5] A similar suggestion is described in [10] that suggests assigning high priorities to processes with low-jitter requirements. The difficulty is similar: processes with low jitter requirements may not have short periods.

can be carefully designed before run-time, the designer has the flexibility to take into account many different possible scenarios in overload situations, and tailor different strategies in alternative schedules to deal with each of them.

(b) A frequently mentioned example of the "flexibility" of the priority scheduling approach, is the fact that there exists a schedulability analysis for the priority scheduling approach that is based only on knowledge of the total processor utilization of the task set. This is supposed to provide more flexibility because "determining the schedulability of a system when an additional task is added requires recomputing only the total schedulability bound and determining whether the new utilization caused by the additional functionality causes the new bound to be exceeded [10]."

What is perhaps less well known about processor utilization based schedulability analyses, is the fact that the use of such analyses may cause the system designer to underutilize the system's capacity. Processor utilization based analyses are invariably pessimistic; they give sufficient but not necessary conditions for schedulability. In other words, if one were to rely on the schedulability analysis, one may be forced to conclude that the fixed priority scheduling algorithm cannot be used, and take measures that further reduce the processor utilization in order to meet the processor utilization conditions provided by the schedulability analysis, even in the simplest of cases where the fixed priority scheduling algorithm may have been able to schedule the processes under the original conditions (for a detailed example, see [17]).

It has also been claimed that, with a pre-run-time scheduling approach, it is more difficult to handle asynchronous processes when compared with using priority scheduling schemes [10] [1][6].

In [17], we provided an example in which the reverse is true. The example shows that, with a pre-run-time scheduling approach, once the pre-run-time schedule has been determined for all the periodic processes, the run-time scheduler can use this knowledge to achieve higher schedulability by scheduling asynchronous processes more efficiently, e.g., it would be possible to completely avoid blocking of a periodic process with a shorter deadline by an asynchronous process with a longer deadline.

In contrast, real-time system designers using the pre-run-time scheduling approach have the freedom to use any optimal scheduling algorithm to construct new pre-run-time schedules that include new processes and add new functionality to the system. Performing modifications to the system on-line is also not difficult. One can easily insert code in pre-run-time schedules that, when activated by an external signal, will cause processor execution to switch from a previously designed pre-run-time schedule to a newly designed pre-run-time schedule during run-time. The system designer is not constrained to use a rigid hierarchy of process priorities, and has more flexibility in designing new pre-run-time schedules to meet changing requirements.

3. It is noted here that sometimes one can achieve higher processor utilization if certain asynchrounous processes with very short deadlines and computation times and long interarrival times are not translated into periodic processes but are kept asynchonous. A complete procedure for doing this, which can be completely automated, is described in [16]. First the procedure determines which asynchronous processes should be translated into periodic processes and which should remain asynchronous based on their respective processor utilizations. Then the procedure reserves processor capacity for processes that are kept asynchronous in the pre-run-time schedule by adding their computation times to the computation times of the periodic processes that have longer deadlines, and allowing those asynchronous processes with very short deadlines to preempt the periodic processes with longer deadlines at run-time. A small run-time scheduler is used for this purpose. With this approach, one would always achieve higher processor utilization compared with any priority scheduling scheme. At the same time, since in most real-time applications the number of asynchronous processes with very short deadlines is normally very small, the complexity will not be significantly increased with this approach and it will still be possible to effectively inspect and verify timing properties of the software.

VIII. Acknowledgements

The author wishes to thank the reviewers for numerous thoughtful comments and helpful suggestions on how to improve this paper. In particular, thoughtful comments from one of the referees, have been incorporated into item (i) and (j) and the last paragraph of section 5 of the paper.

References

[1] N. Audsley, K. Tindell and A. Burns, "The end of the line for static cyclic scheduling," *Proc. Fifth Euromicro Workshop on Real-Time Systems*, 36-41, 1993.

[2] R. Alur, D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, 126, 1994, 183-235.

[3] A. Burns, K. Tindell, and A. Wellings, "Effective analysis for engineering real-time fixed priority schedulers," *IEEE Trans. Software Eng.*, 21, 1995, 475-480.

[4] E. W. Dijkstra, "Cooperating Sequential Processes." In *Programming Languages,* F. Genuys, Ed. Academic Press, 1968, 43-112.

[5] E. W. Dijkstra, "Structured Programming", in *Software Engineering Techniques*, J. N. Buxton and B. Randell, Ed. Brussels, Belgium; NATO Sci. Affairs Div. 1970, 84-87.

[6] B. Dutertre, "Formal analysis of the priority ceiling protocol,"

[6] In [10], where a particularly rigid version of a pre-run-time scheduling approach, the cyclic executive, was applied to an example problem, in order to show the difficulties in applying the cyclic executive, the author did not illustrate how the fixed priority executive could solve the same example scheduling problem. The fixed priority executive will have an equal or even greater difficulty in handling that same example problem. In the other paper [1] which attempts to show that a priority scheduling approach can solve an example problem that was given in one of our papers, the example problem parameters in our paper [21] were changed. If the original problem parameters in our paper were used, the proposed solution would fail. In addition, the proposed solution used offsets for which apparently no algorithm was given that can systematically compute those offsets.

IEEE Real-Time Systems Symposium, Orlando, FL, pp. 151-160, Nov. 2000.

[7] E. A. Emerson, and E. M. Clarke, "Using branching time temporal logic to synthesize synchronization skeletons," *Science of Computer Programming*, vol. 2, 241-266, 1982.

[8] T. Henzinger, Z. Manna, and A. Pneuli, "Temporal proof methodologies for real-time systems," *Proc. 18th ACM Symp. on Principles of Programming Languages*, 353-366, 1991.

[9] F. Jahanian and A. Mok, "A graph-theoretical approach for timing analysis and its implementation," *IEEE Trans. on Computers*, vol. 36, pp. 961-975, 1987.

[10] C. D. Locke, "Software architecture for hard real-time applications: cyclic executives vs. fixed priority executives," *Journal of Real-Time Systems*, 4, 37–53, 1992.

[11] A. K. Mok, "Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment", Ph.D Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1983.

[12] A. K. Mok, "The design of real-time programming systems based on process models", in *Proc. IEEE Real-Time Systems Symposium*, pp. 5-17, Dec. 1984.

[13] J. Ostroff, "Temporal Logic of Real-Time Systems," Research Studies Press, 1990.

[14] D. L. Parnas, "On the criteria used in decomposing systems into modules," *Commun. ACM*, vol. 15, pp. 1053-1058, Dec. 1972.

[15] D. L. Parnas, "Inspection of Safety Critical Software using Function Tables", Proc. IFIP World Congress 1994, vol. 3, August 1994.

[16] J. Xu and Kam-yiu Lam, "Integrating run-time scheduling and pre-run-time scheduling of real-time processes." Proc. *23rd IFAC/IFIP Workshop on Real-Time Programming*, Shantou, China, June 1998.

[17] J. Xu and D. L. Parnas, "Fixed priority scheduling versus pre-run-time scheduling." *Real-Time Systems,* 18, pp. 7-23, Jan. 2000.

[18] J. Xu and D.L. Parnas, "On Satisfying Timing Constraints in Hard-Real-Time Systems." *IEEE Trans. on Software Engineering*, vol. 19, pp. 1-17, Jan. 1993.

[19] J. Xu, "Multiprocessor scheduling of processes with release times, deadlines, precedence, and exclusion relations," *IEEE Trans. on Software Engineering*, vol. 19, Feb. 1993.

[20] J. Xu and D.L. Parnas, "Pre-run-time scheduling of processes with exclusion relations on nested or overlapping critical sections." Proc. *Eleventh Annual IEEE International Phoenix Conference on Computers and Communications (IPCCC-92),* Scottsdale, Arizona, April 1-3, 1992.

[21] J. Xu and D. L. Parnas, "Scheduling processes with release times, deadlines, precedence, and exclusion relations," *IEEE Trans. on Software Engineering*, vol. 16, pp. 360-369, Mar. 1990. Reprinted in "Advances in Real-Time Systems," edited by J. A. Stankovic, and K. Ramamrithan, IEEE Computer Society Press, pp. 140-149, 1993.