Software Restructuring to Improve Schedulability

Rachid Boudour*, Mohamed T. Kimour**
Department of Computer Science,
LRI, University of Annaba
Bp. 12, Annaba, Algeria
Algeria
Tel / Fax: 213-38-87-27-56

Abstract: Real-time programs must be logically correct and must complete without timing errors. Errors due to overloaded resources are exposed very late in a development process, and often at run-time. Specifically, when a set of tasks are found to be overloaded and not schedulable, the developers usually remedied these problems by manual-intensive operations such as instrumentation, measurement, code tuning and eventually redesign. Such operations are slow and uncontrollable. We propose a semi-automatic alternative to this manual process, based on the task restructuring. Our approach is an extension of the one proposed by R. Gerber that only treats some monolithic and simple programs. It consists for the scheduling perspective, in restructuring real-time programs, which can be complex and contain a collection of procedures. This restructuring, by using the interprocedural program slicing concept and a finer dependence model, allows to make flexible the deadline task and therefore, to improve its schedulability.

Key-Words: Program Slicing, Real-time Constructs, Timing Constraints, Behavior Model, Task Scheduling.

1 Introduction

Real-time computer applications are increasingly being designed and used in every life. Moreover, these systems are often parts of critical applications. Typical real-time applications are control systems (manufacturing systems, robotics), monitoring systems (patient monitoring, air traffic) and communication systems. The correctness of a real-time system depends not only on how concurrent processes interact, but also on the time at which these interactions occur. In these systems, tasks are regularly performed to control physical processes and monitor functioning. They sample information to process in order to produce commands for actuators. Invocation of these typical tasks occurs periodically, every **T** time units. The most common requirement is that a task invocation must be completed within **D** time units after it is ready. The parameters **T** and **D** are called respectively the *period* and the *deadline*.

Either a failure to perform a task at the appropriate time or a flaw in the control program's logic can yield catastrophic consequences in hard real-time systems. Thus, meeting the timing constraints is extremely important in such systems. To satisfy their deadlines, tasks must be appropriately scheduled. Algorithms for scheduling tasks in real-time systems typically assume that their worst-case execution times are known. Such a system is designed to ensure that all tasks can complete by their deadlines as long as no task in a system executes longer than its worst-case execution time. A task which overruns may lead to missed deadline and failure of the whole system.

On the other hand, errors due to overloaded resources are exposed very late in a development process, and often at run-time. Specifically, any deadline violation makes tasks unschedulable; to realize schedulability, programmers usually remedied to these problems by manual-intensive operations such as instrumentation, measurement, code tuning and restructuring. Such a manual and costly process is slow and uncontrollable [1, 2]. In certain cases, entire subsystems may have to be altogether redesigned.

In this paper, we present a semi-automatic alternative to this process, based on two concepts: interprocedural slicing [3] and fixed priorities scheduling [4, 5]. A slice of program, with regard to a point p and a variable v, consists of statements of the program that can affect the value of v, at the point p [3]. Its behavior is identical to the initial program with regard to a given criterion. We can for example, isolate a slice with regard to a critical-instruction in the program, an instruction, which emits command to an actuator or an instruction of alarm in a control task.

While respecting the different dependencies in the code of an unschedulable task, we ensure so that statements that influence the deadline are executed in the first place.

The fragment of code that corresponds to a local computation can be deffered. This operation permits us to defer the execution of a part of the code, while preserving the initial semantic.

In the scheduling perspective, Gerber and Hong [2] used program slicing to divide a program task. In a different context (the one of the maintenance), Gallagher and Lyles used slices in program decomposition [6]. However, the presented approaches treat only monolithic and simple programs. They are very restraining with regard to the reality of real-time system, where we often find structures of programs that contain procedures [14].

In contrast to these works [6, 2] and while developing a finer dependence model of program behavior, slices that we extract from the program can contain a collection of procedures. Furthermore, they are executable and precise. Since our transformed tasks are sequentially executed, all slicing decisions are taken statically before the application run-time. Such a static transformation type preserves the temporal determinism, which is essential in real-time systems.

These interprocedural slices are the basis of our method of real-time program restructuring. Thus, we can restructure a program automatically by the displacement of its nontemporal fragment after the one (temporal slice) that directly influences the deadline. This process will help the programmer to achieve schedulability and to get a globally correct application.

In the next section, we show how program dependencies are represented, and how to use this representation to extract temporal slices. In section 3, we presented our restructuring algorithms. Section 4 presents the tool of Real-Time Task Restructuring (RTTR) that we conceived and implemented. Finally, the conclusion is presented in section 5.

Task	Ti	Ci	Di
τ_1	80	20	80
τ_2	150	80	150
τ_3	200	35	200

Table 1 Example of three tasks

Schedulability of τ_2
S0=1(20)+1(80)=100 ms
S1=2(20)+1(80)=120 ms
S2=2(20)+1(80)=120 ms

Table 2 Analysis of τ_2

Schedulability of τ_3
S0=1(20)+1(80)+1(35)=135 ms
S1=2(20)+1(80)+1(35)=155 ms
S2=2(20)+2(80)+1(35)=235 ms

Table 3 Analysis of τ_3

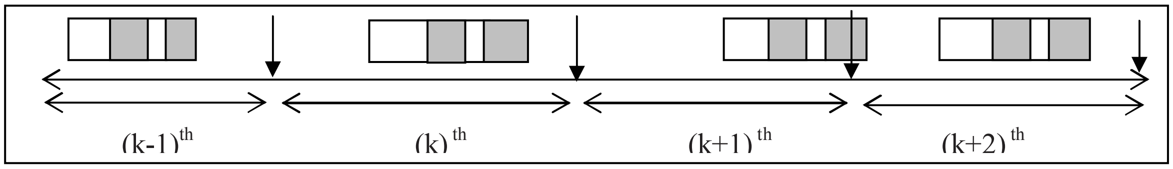

Fig.1a Execution behavior of the periodic task *before* restructuring.

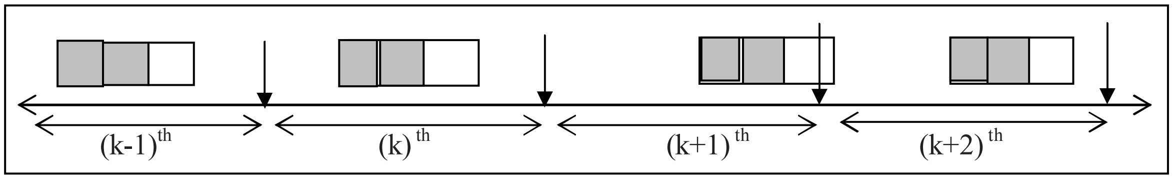

Fig.1b Execution behavior of the periodic task *after* restructuring.

To show how we analyze the task set schedulability, consider the case of three tasks τ_1, τ_2, and τ_3. Their characteristics are described in table 1. The line rank corresponds to the task priority. By applying of the above equations, for τ_1, S_0=20, S_1=20, S_0=S_1, the worst-case completion time is equal to 20 ms (□80 ms), this task is then schedulable. For the task τ_2, the schedulability analysis (table 2) shows that it is schedulable. Its worst-case completion time is equal to 120 ms (□150 ms), while the schedulability of the task τ_3 (table 3) is not guaranteed, since its worst-case completion time is equal to 235 ms (□ 200 ms).

2 Task Restructuring

Our objective is to obtain flexible deadlines of tasks, because the schedulability is improved as one increases the flexibility of deadlines. This effect can result from the fact that one allows a task to go beyond of its deadline [15]. We can get this advantage by task restructuring to obtain two fragments. The first is a temporal fragment (noted FT) that must satisfy the deadline. The second is nontemporal fragment (noted FnT), which corresponds to a local computation that can tolerate a certain delay. Then, fragments will be sequentially recomposed, to constitute a new program of the task in question, which is semantically equivalent to the original task program.

Fig.1a represents the execution behavior of a periodic task. Hatched Parts in figures 1a and 1b represent temporal components of the task program. At the (k+1)th period, the execution of this task goes beyond its deadline and becomes thus, unschedulable (fig.1a). However, in the case where temporal statements would complete within the prescribed period, the execution of the whole task is acceptable (fig.1b). Our restructuring approach uses the interprocedural slicing that is a fundamental operation of program behavior analysis [3, 6].

On the other hand Program slicing [3] is a technique for automatically decomposing high-level programs by analyzing its control and data flow information. Static slicing techniques aim at extracting a minimal program that captures the behavior of a source-code program with respect to a specified variable and location; this problem is solved by a reachability analysis. Thus, we need an explicit representation of data and control flows. This is made by a Dependence Graph, which reduces program slicing into a graph walk problem [7]. A slice is computed by backward traversing the control and data dependence edges of the graph. The slice corresponds to the subgraph containing the reached vertices and edges. A monolithic program (program without procedure calls) is represented by a Procedure Dependence Graph (PDG). A program containing collection of procedures is represented by a System Dependence Graph (SDG) [8, 9].

To represent the timing constraints, we propose by the analogy of the program dependence graph, to implement a graphical model, which consists of a set of vertices and edges. Vertices are associated with instructions or variables, while edges are associated with dependencies and real-time constraints [13].

An instruction can be classified as an *externally viewed instruction* (EVI) or *an internally viewed instruction* (IVI) based on the effect of the instruction. The effect of IVI is limited to the internal computation, while an EVI changes the status of the control environment. For example, the variables defined as *volatile* in the C language [1] or commands to control robot or actuators are EVIs, and their execution must meet the timing constraints specified. Because EVIs may depend on values computed by IVIs, any such dependence also implies a relative timing constraint (execution order) which must be preserved.

2.1 Dependence Representation

A PDG is composed of vertices and edges representing respectively program statements and their dependencies. With the exception of call statements, a single vertex represents assignment statements, input statements and output statements, and the predicate of conditional (if) and while-loop statements. Additionally, there is a distinguished vertex called the *entry* vertex, and an *initial-definition* vertex for each variable that may be used before being defined. The source of a *control dependence* edge is either the *entry* vertex, a *predicate* vertex, or a *call* vertex. Each edge is labeled either true or false. A control dependence edge, from vertex *v* to vertex *u,* means that during execution, whenever the predicate represented by *v* is evaluated and its value matches the label on the edge to *u*, then the program component represented by *u* will eventually be executed, provided the system terminates normally.

A data dependence edge from vertex *v* to vertex *u* means that the system's behavior might change if the relative order of the components represent by *v* and *u* were reversed. There are two kinds of data dependence edges, *flow dependence* edges and *def-order dependence* edges. A flow dependence edge connects vertex *v* that represents an assignment to a variable *x* to a vertex *u* that represents a use of reached by that assignment. A def-order edge runs between two vertices, *v* and *u* that both represent assignments to variable *x* where both assignments reach a common use, and *v* lexically precedes *u*.

To represent the multi-procedural program dependencies, we use a System Dependence Graph (SDG) [8]. It is composed by set of PDG, connected by edges that symbolize dependencies of *calls* and *summaries* (Figure 2). Summary edges represent *transitive* dependence due to the effects of procedure calls. A call statement is represented using a call vertex and four kinds of parameter vertices that represent parameter passing. On the calling side, parameter passing is represented by *actual-in* and *actual-out* vertices, which are control dependent on the *call* vertex. In the called procedure, parameter passing is represented by *formal-in* and *formal-out* vertices, which are control dependent on the procedure's entry vertex. *Actual-in* and *formal-in* vertices are included for every parameter and global variable that may be used or modified as a result of the call. *Formal-out* and *actual-out* vertices are included for every parameter and global variable that may be modified as a result of the call.

To address the calling context problem, the system dependence graph includes *summary edges*. A summary edge connects actual-in vertex *v* to actual-out vertex *u* if there a path in the SDG from *v* to *u* that respects calling context by matching calls with returns. We add three kinds of interprocedural edges: (1) A call edge connects each call vertex to the corresponding *procedure-entry* vertex. (2) *Parameter-in* edge connects each *actual-in* vertex at a call site to the corresponding *formal-in* vertex in the called procedure. (3) *Parameter-out* edge connects each *formal-out* vertex to the corresponding *actual-out* vertex at each call site on the program. The global variables are generally treated as supplementary actual parameters.

In a perspective of restructuring for scheduling, we are concerned by imperative programming languages, containing real-time extensions. The program model that we used assumes the following properties: (1) a complete system consists in a main program (Main) and a collection of procedures, (2) parameters are passed by value-results.

2.2 Interprocedural Slicing

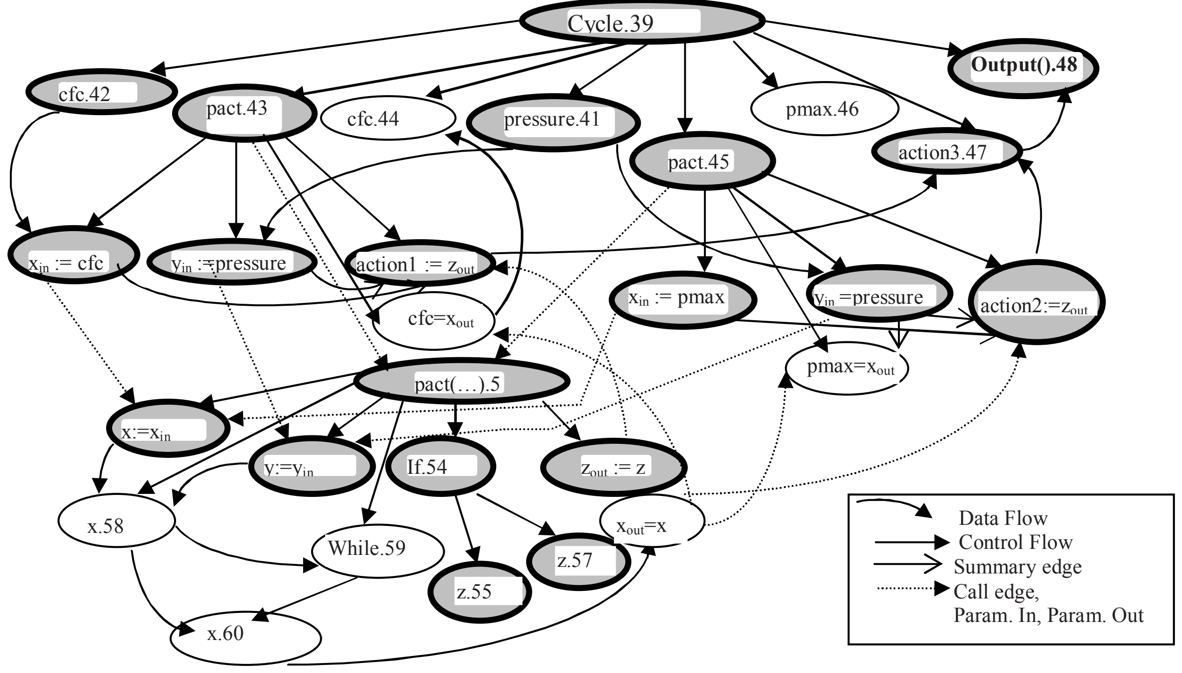

Fig.2 *SDG of the task program.*

Furthermore, to obtain precise and executable slices, we use an adapted and fine dependence model where (1) the element is an action on variable, (2) timing constraints are represented [13], and (3) slices are relative to several procedures. Our objective is obviously to produce the temporal component, represented by a precise and executable slice. The temporal component incorporates all instructions that influence a given criterion. The criterion is defined by line number and a variable. The line number can reference an output instruction or command transfer to an actuator, for example. This instruction corresponds to an EVI type [1, 15].

On the other hand, an interprocedural slice of an SDG with respect to the vertex *v* is computed using two passes over the graph [16]. Summary edges are used to permit «moving across» a call site without having to descend into the called procedure; thus these is not need to keep track of calling context explicitly to ensure legal execution paths are traversed. Both passes operate on the SDG, traversing edges to find the set of vertices that can reach a given set of vertices along certain kinds of edges.

The traversal in Pass 1 starts from all vertex *v* and goes backward (from target to source) along flow edges, control edges, call edges, summary edges, and parameter-in edges, but not along def-order or parameter-out edges. The traversal in Pass 2 starts from all vertices reached in Pass 1 and goes backward along, flow edges, control edges, summary edges, and parameter-out edges, but not def-order, call, or parameter-in edges. The result of an interprocedural slice consists of the sets of vertices encountered during Pass 1 and pass 2 [16, 17].

3 Restructuring Algorithms

Consider a real-time program called M that contains calls of a procedure P. SDG can be constructed automatically from the program [8]. We propose three algorithms to restructure a given task source code.

Algorithm 3.1 produces the temporal slice with regard a given criterion. To do this, it performs pass 1 and pass 2 treatment, described in the above section. It marks vertices corresponding to temporal statements (including procedure calls) of the main program (M) and of procedures. The silce obtained at his step can be nonexecutable. To make it executable, we perform the following actions: (1) add vertices that remove actual-in vertex mismatches; (2) add vertices that remove actual-out vertex mismatches; and (3) produce a system from the resulting set of vertices and the original system. This algorithm also marks every code line corresponding to a marked vertex in the SDG. Hatched Vertices in Fig.2, represent the temporal slice, obtained by application of our algorithm 3.1 on the example (Figure 3.a) with regard to the criterion {<48>,<action3>}.

Algorithm 3.2 uses the result of the algorithm 3.1 (treated SDG, marked code and procedures). In the program M, we erase all unmarked lines so that only temporal statements remain. We perform the same treatment on the procedure P, to get a temporal subprocedure that we call P1.

Algorithm 3.3 uses the result of the algorithm 3.1 (treated SDG, marked code and procedures) to compute the nontemporal fragment. Consider our example of real-time program, called M. From the main program, we erase lines that correspond to marked vertices, except, the conditional statements, the while and the call of procedure.

```
ALGORITHM 3.1. Identifies executable slice T(SDG,n). (n: Slicing criterion).
  Step 1, Start from n and go backward (from target to source) along flow edges, control edges, call edges,
      summary edges, and parameter-in edges, but not along def-order or parameter-out edges.
  Step 2, Start from every vertices reached in Step 1 and go backward along flow edges, control edges,
      summary edges, and parameter-out edges, but not along def-order, call, or parameter-in edges.
  Step 3, While there exists an actual-in vertex y, which mismatches with marked formal-in vertex, mark the
      vertex y. Perform the same treatment for the actual-out vertex. Mark all vertices (unmarked again)
      which depend (flow dependence) on vertices computed at step 2.
```

```
ALGORITHM 3.2. Compute the temporal fragment FT(treated SDG, Task Program).
  Step 1, From the program M and every procedure P, erase lines corresponding to vertices not marked in
      the SDG, by the algorithm 3.1.
  Step 2, Replace each call of P, by a call to P1.
The result represents the FT fragment.
```

```
ALGORITHM 3.3. Compute the nontemporal fragment FnT (treated SDG, task program).
  Step 1, Do steps 2 & 3, for all S representing the main program and every procedure marked by the
      algorithm 3.1.
  Step 2, Erase from S, lines which are marked by the algorithm 3.1, except lines of conditional instruction
      and call of procedure.
  Step 3, Erase lines of conditional statements and calls of procedure which control an empty instruction set.
  Step 4, Replace every marked call of the procedure P, by a call of the subprocedure P2, and every no
      marked call of the procedure P, by a call of P1 followed by a call of P2.
The result represents the FnT fragment.
```

We perform the same treatment for the procedure P, to get a nontemporal subprocedures (P2). Because the procedure P is transformed in two subprocedures P1 and P2 (Figure 3), we replace then, every marked call of P in the remainder of the program M, by a call of P2.

In order to only having two declarations for the procedure P, we replace every unmarked call of P, by a call of P1 followed by a call of P2. If P is not marked at all, we conserve this procedure in the non-temporal fragment. Figure 4b shows the two subtasks (FT and FnT). They are obtained by applying the algorithms 3.1, 3.2 and 3.3 on our example (Figure 4a), and its SDG (Figure 2), with regard to criterion {<48 >, <action3>}. The above program fragment (Figure 4a) is inspired from the work of T. M. Chung and H. G. Dietz [1]. The code of task τ_3, embodies timing constraint, and it could easily have been found in a real application.

The construct *cycle(200)* in line 39 specifies the task period. Thus, the instructions in the block enclosed by {and} must be completed within 200 *ms*. The program fragment contains the code of the task τ_3. This task performs the following actions, *every 200 ms*:

- Read a new measure on the pressure sensor (line 41);

Using the new reading data (pressure) and the current state (values of cfc and pmax),

- Produce an actuator command (action at line 47) to reduce pressure.
- Update the state;
- Send command to the actuator before taking the next sensor reading (line 48).

```
1:   main()
20:        •••
39:     cycle: (200) /* task   period*/
40:     {
41:     input(sensorP, pressure);
42:     cfc=cfc+pmax/3;
43:     pact(cfc, pressure, action1);
44:     cfc=cfc-pmax/100;
45:     pact(pmax,pressure,action2);
46:     pmax=100;
47:     action3=F(action1,action2);
48:     output(actuator, action3);
49:     }
50:     pact(x, y, z);
51:         float x, y;
52:         char z;
53:         {
54:     if (x>y)
55:       z=«cmdpress1»;
56:     else
57:       z=«cmdpress2»;
58:     x=x+y;
59:     while (x>y)
60:             x=x-0.03;
61:   }
```

Fig.4a Original Task Program

```
1:  main()
20:        •••
39:     cycle: (200)           /* task  period*/
40:     {
41:     input(sensorP, pressure);
42:     cfc=cfc+pmax/3;
43':     pact1(cfc, pressure, action1);
45':     pact1(pmax,pressure,action2);
47:     action3=F(action1,action2);
48:     output(actuator, action3);
43»:   pact2(cfc, pressure);
44:     cfc=cfc-pmax/100;
45»:   pact2(pmax, pressure);
46:     pmax=100;
49:     }
50':     pact1(x, y, z);
51':         float x, y;
52':         char z;
53':        {
54:     if (x>y)
55:       z=«cmdpress1»;
56:     else
57:       z=«cmdpress2»;
61':   }
50»:    pact2(x, y)
51»:        float x, y;
53»:        {
58:         x=x+y;
59:         while (x>y)
60:             x=x-0.03;
61»:   }
```

Fig.4b Restructured Task Program

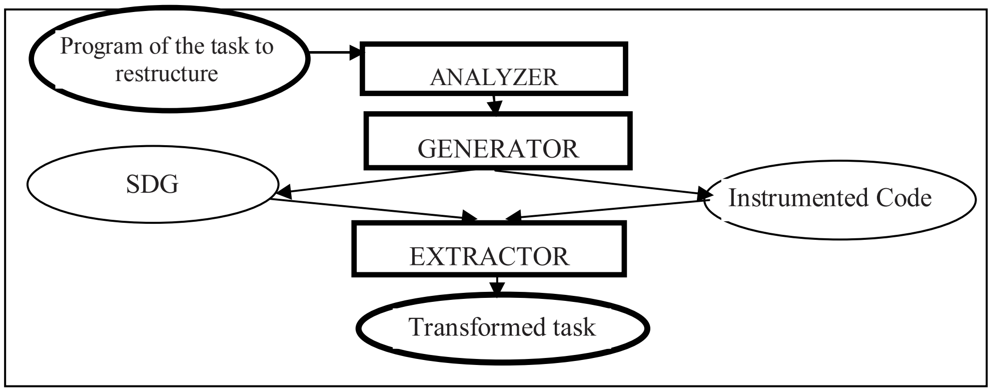

Fig. 5 RTTR Structure.

4 Restructuring Tool

We have conceived and implemented a tool (Figure 5) of task restructuring, called RTTR (Real Time Task Restructuring). Based on the above-mentioned techniques, it has been developed on a processor Pentium-II, using the Visual C++ language. It is composed of three modules: the Analyzer, the Generator and the Extractor. From a given real-time program, the Analyzer performs a lexical, syntactic and semantic analysis to produces an instrumented code. From the instrumented code, which is a well-structured code, it constructs the SDG. The *Generator* module visualizes the instrumented source code (where each instruction is labeled by a single number (IdDep)) and gives measurements. The Extractor is exploited in interactive fashion. It offers to the user the means to extract the two fragments (FT & FnT) as well as the code of the task transformed.

5 Conclusion

In this paper, we presented a static approach to improve the real-time task schedulability. It is based on interprocedural program slices and fixed priorities scheduling techniques. Program slicing is a technique for restricting the behavior of a program to some specified subset of interest. Thus we have used slices to decompose a task into two fragments. The proposed approach, based on a fine representation model of functional and temporal program behavior, allows us to automatically compute a precise and executable slice, by analyzing data and control flow. Furthermore, since the transformations modify the instruction order, our modified slicing algorithm takes into account the outputs, the anti-dependencies and the flow dependencies. We note that we do not dynamically reconstruct the outputs of transformed tasks, because the generated output instructions are in only one slice and the local computation instructions are in another slice. By determining the timed-slice of a task, we have shown that we can restructure an initially unschedulable application to another one, which will be equivalent but schedulable.

On the other hand, we have used the rate-monotonic scheduling algorithm, since it is among the best-known algorithms and more used by the real-time practitioners. Its analytical model allows performing a precisely temporal analysis, and it is simple to implement. Moreover, the traditional real-time model, while very simple, has the reference point for most advances in the area of real-time systems. It has been generalized to accommodate distributed systems, network protocols, shared databases, etc.

To focus on the real time, we have treated in this article, constructs that are more used in real-time programming languages. Currently, we are studying the more complex temporal constructs and other constraints of scheduling techniques, such as precedence constraints between tasks and resource constraints.

References

[1] T.M. Chung, H.G. Dietz. Language Constructs and Transformations for Hard Real-Time Systems, ACM SIGPLAN Notices, Vol.30, No11, November 1995.

[2] R. Gerber, S. Hong. *Slicing Real-Time Programs for Enhanced Schedulability*. ACM Trans. on Prog. Lang. and Sys. vol.19, n°3, May 1997, pp.525-555.

[3] M. Weiser. *Program Slicing*. IEEE Trans. on Soft. Eng. vol.10, n°4, July 84, pp 352-357.

[4] C.L. Liu, J.W. Layland. *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*. ACM, vol.20, n°1, January 1973, pp.46-61.

[5] M. Klein, J. Lehoczky, R. Rajkumar. *Rate-Monotonic Analysis for Real-Time Industrial Computing*, IEEE Computer, vol.27, n°1. Jan 1994.

[6] K. Gallagher and J. Lyle, *Using Program Slicing in Software Maintenance*, IEEE Transactions on Software Engineering, vol. 17, n° 8, p.751-761, 1991.

[7] K. Ottenstein, L. Ottenstein. *The Program Dependence Graph in a Software DevelopmentEnvironment*. ACM SIGSOFT/SIGPLAN, ACM Press, New York, 1984, pp.177-184.

[8] D. Binkley. *Precise Executable Interprocedural Slices*. ACM Letters on Prog. and Sys. vol.2, n°1-4, 1993, pp.31-45.

[9] M.S. Bendelloul, Z.E. Bouras, S. Ghoul, T. Khammaci. *Assistance à la Compréhension de Programme. Un Modèle et un Algorithme de Fragmentation*. Génie Logiciel, n°45, Sep. 1997, pp.32-42.

[10] A. Burns, A. Wellings. *Real-Time Systems and their Programming Language.* Addison-Wesley Publishing Company, Inc., Wokingham, England.1990.

[11] A. Burns, K. Tindell, A. Wellings. *Effective Analysis for Engineering Real-Time Fixed Priority Scheduling.* IEEE Trans. Soft. Eng. Vol.21, No.5,.May 1995, pp.475-480.

[12] T. Hamalainen, J. Siltanen, A Viinikainen, J. Joutsensalo, Adaptive Tuning of Scheduling Parameters, *WSEAS Transactions on Computers*, Vol.2, No. 1, January 2003.

[13] B. Dasarathy. Timing Constraints of Real-Time Systems: Constructs for expressing them, Method for Validating them. *IEEE Trans. On Soft. Eng;* Vol.11, No 1, 1985, pp.80-86.

[14] S. Tadamea, Y. Kishimoto, Program Code Analysis Focused on its Structure, *WSEAS Transactions on Computers*, Vol.2, No. 1, January 2003.

[15] S. Hong, R. Gerber. *Compiling Real-Time Programs into Schedulable Code*. ACM SIGPLAN Notices. Vol.28, No 26. June 1993. pp.166-176.

[16] S. Horwitz, T. Reps, D. Binkley. *Interprocedural Slicing using Dependence Graph*. ACM trans. Program. Lang. Syst. Vol.12, No 1, January 1990, pp.26-60.

[17] F. TIP. *A Survey of Programming Slicing Techniques*. J. Program. Lang. Vol.3, No3, 1995, pp.121-189.