BULGARIAN ACADEMY OF SCIENCES

CYBERNETICS AND INFORMATION TECHNOLOGIES • Volume **16**, No 2
Sofia • 2016 Print ISSN: 1311-9702; Online ISSN: 1314-4081
DOI: 10.1515/cait-2016-0022

Multi-Partite Graphs and Verification of Software Applications for Real-Time Systems

Victor Nikiforov[1], *Sergey Baranov*[2]

[1]*St. Petersburg Institute for Informatics and Automation of the Russian Academy of Sciences, 14 liniya, 39, V. O., St. Petersburg, 199178, Russia*
[2]*ITMO University, av. Kronverkski, 49, St. Petersburg, Russia*
Emails: nik@iias.spb.su SNBaranov@gmail.com
http://www.spiiras.nw.ru/index.php?newlang=english

Abstract: *Aspects of static verification of software applications for real-time systems are considered. A verification method based on oriented multipartite graphs is suggested for checking whether mutual blockings (deadlocks or clinches) could occur in a real-time multitask application and estimate the duration of high-priority task blocking by lower-priority tasks due to the application structure.*

Keywords: *Real-time systems, multi-task software applications, multi-core processors, feasibility analysis, shared resources, blocking factor.*

1. Introduction

The characteristic feature of software real-time applications is that they work in the "timing structure" determined by external processes flow. Certain time frames of various flexibility levels for information exchanges between Real-Time (RT) software application components and external processes provide a sort of conformity between their flows.

The required conformity between the timing structure, which reflects the external process flow and that for execution of the software components, is a manifestation of a general requirement of the architecture of a reasonably constructed RT system – observing the principle of structured conformity. In its most general form this principle reads: for a reasonably constructed Real-Time System (RTS) the break-down structure of software objects complex should be a mapping of the structure of the set of external objects [1]. This principle guides the development of the architecture of a real-time software application. A major corollary of this principle is the requirement to organize a software application in the form of a complex of cooperating tasks.

Logical correctness of multi-task software applications. Cooperating tasks coordinate their activities by exchanging data and synchronizing (signalling) informative messages and by sharing common system resources: computing (processors and cores of multi-core processors) and informational (global data arrays, interface registers of peripheral devices, elements of man-machine interface, etc.) Aspects of logical correctness should be addressed among other issues in the process of verification of multi-tasking software complexes (when developing real-time systems as well as simulation systems, etc.). Particular issues are assuring integrity of shared informational resources and preventing incorrect situations when tasks interrelated by operations of accepting signalling messages are trapped in the state of infinite waiting (clinches and deadlocks).

Dynamic correctness of software applications for RTS. When developing a real-time system, verification aspects related to *dynamic* correctness, which include feasibility assurance – on-time execution of tasks within the software application [2, 3], should be addressed as well, along with aspects of logical correctness.

Developing software applications for real-time systems, one should resolve conflicts between requirements to reduce the size of allocated computing resources and those to ensure feasibility of particular tasks and the software application as a whole. Resolving this conflict allows to increase the efficiency of computer resources usage while preserving the dynamic correctness of the real-time system.

2. Tasks and jobs

To verify software applications for real-time systems parametric and structural characteristics of static and dynamic application components (tasks and jobs) shall be used, which are deliberately aimed at resolving the abovementioned problems of logical and dynamic correctness.

The *j*-th activation of the task τ_i means arrival of its *j*-th instance – arrival of the job $\tau_i^{(j)}$ – some new dynamic component of application. Job arrival means an increase of the number of competitors for the processor time and, therefore, a change of conditions for processor time allocation. A change in conditions for system resources allocation being called a *system event* in general, job arrival should be considered as its instantiation. Job termination – a decrease of the number of competitors for the processor time – is another example of a system event.

The behaviour of each job $\tau_i^{(j)}$ shall be limited by static parameters of the corresponding task τ_i:

- the task weight C_i, that is the maximal processor time required for a single execution of the task τ_i ;
- the task period T_i, that is the length of the time interval between two successive activations of the task τ_i (the length of the time interval between the moments $t_{arr}(\tau_i^{(j)})$ and $t_{arr}(\tau_i^{(j+1)})$ of arrivals of jobs $\tau_i^{(j)}$ and $\tau_i^{(j+1)}$ respectively);
- the task deadline D_i, that is the maximal acceptable length of time interval between the job $\tau_i^{(j)}$ arrival time $t_{arr}(\tau_i^{(j)})$ and its termination time $t_{end}(\tau_i^{(j)})$.

The used *scheduling mode* (or, simply, *scheduling*) determines the ordering in which processor time is allocated to serve active jobs [2]. Scheduling may be

defined as a means to match jobs with integer-valued *priorities*: at the moment of each system event active jobs are assigned with certain priority values; the resource of processor time is allocated to active jobs of the highest priorities. In most RTS, priorities are assigned statically. Let's enumerate tasks τ_1, τ_2, ..., τ_n, which compose an application, according to the descending ordering of their priorities: τ_1 is a task with the highest priority, and τ_n is a task with the lowest priority. The most widely used scheduling with static priorities is that of RM (Rate Monotonic), where task priorities decrease as the values T_i increase.

Expected timelines of jobs in an RTS should be guaranteed: the duration $r_i^{(j)} = t_{\text{end}}(\tau_i^{(j)}) - t_{\text{arr}}(\tau_i^{(j)})$ of the interval of existence of any instance $\tau_i^{(j)}$ of the task τ_i should not exceed the maximal acceptable value D_i (the task τ_i *deadline*). Using the notion of *response time* $R_i = \max\{r_i^{(j)} | \ j=1, 2, ...\}$ of the task τ_i, the feasibility requirement for the task τ_i (guaranteed timeliness of execution of any of its instances) may be formulated as the inequality: $\forall i \ D_i \geq R_i$.

Most actual RTS applications contain *interdependent* tasks which may fall into states of waiting for signal messages from other tasks. This means that the interval of existence of the job $\tau_i^{(j)}$, which accepts a signal message, may contain subintervals where the processor is busy with execution of jobs with priorities lower than that of $\tau_i^{(j)}$. Such subintervals are called *blocking* intervals of the job $\tau_i^{(j)}$, within such interval the job $\tau_i^{(j)}$ is in a *blocking state* (is waiting for incoming particular signal messages). The total duration $b_i^{(j)}$ when the job $\tau_i^{(j)}$ is in blocking states contributes to the value $r_i^{(j)}$ of the response time of this job. The maximal possible contribution $B_i = \max\{b_i^{(j)} | \ j=1, 2, ...\}$ for jobs of the type τ_i is called a *blocking factor* of the task τ_i. For *independent* task τ_i blocking factor is absent ($B_i = 0$).

Another factor contributing to the value $r_i^{(j)}$ is the *preemption time* – the total time $p_i^{(j)}$ when the job $\tau_i^{(j)}$ was waiting for the resource of the processor busy by other higher priority jobs. The maximal such duration $I_i = \max\{p_i^{(j)} | \ j=1, 2, ...\}$ among jobs of the type τ_i is called a *priority factor* of the task τ_i.

Exact estimation of the B_i value requires taking into account the task internal structure, that may by presented by route networks [4]. Fig. 1 represents expressive means of a particular variant of route networks.

Synchronizing interface elements of the mutex type are used as a means to control access to shared information resources. For each such resource g a separate mutex `mut` is formed. Let mutex `mut_1` be formed to control access to the shared resource g_1. Then each program code segment within which the resource g_1 is accessed (each critical interval for accessing the resource g_1) is framed with operators on the mutex `mut_1`: it starts with the operator `lock(mut_1)` – lock the mutex `mut_1`; and ends with the operator `unlock(mut_1)` – unlock the mutex `mut_1`. If at the moment of invoking the operator `lock(mut)` the mutex `mut` is in the state "locked", then job execution is suspended until this resource becomes available by the operation `unlock(mut)` within the job which currently owns this resource. Operators `lock/unlock` break down the task code into segments.

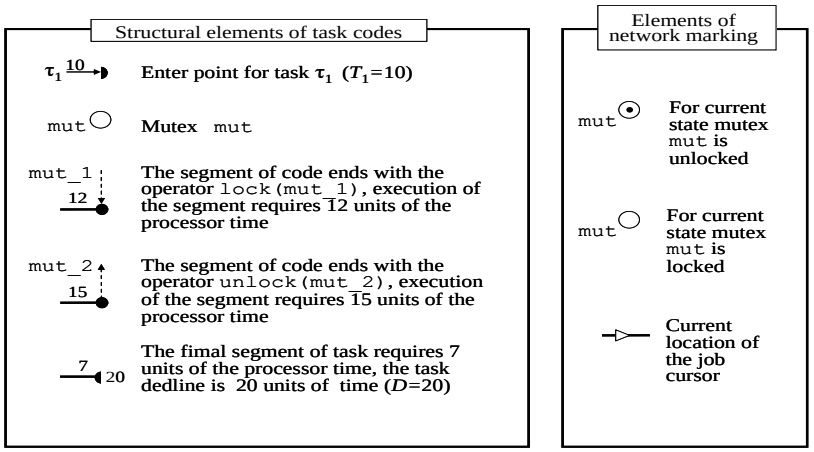

Fig. 1. Elements of route networks

Fig. 2 provides an example of a route network consisting of two tasks. Elements "task entry point" are marked with the task names τ_1 and τ_2 with the values of the task period (T_1=20, T_2=32). The ending segment provides the maximal value for the task execution (D_1=20, D_2=32). The number of processor time units needed for task execution is specified for each segment.

The first segment of the task τ_1 ends with the operator `lock(mut_1)`, the third segment of the task τ_1 ends with the operator `unlock(mut_1)`: the piece of code between these two operators is a *critical interval* for access to the resource controlled by mutex `mut_1`. The length of this critical interval is equal to 4 units of the processor time (the sum of the lengths of the second and the third code segments of the task τ_1). The length of the critical interval of the task τ_1, which corresponds to the mutex `mut_2`, equals 5. Critical intervals of the task τ_1 with respect to `mut_1` and `mut_2` overlap at the third task code segment. The critical interval of the task τ_2 with respect to `mut_1` is nested into the critical interval with respect to `mut_2`. Tasks τ_1 and τ_2 are interdependent, there instances may be trapped in the "waiting" state.

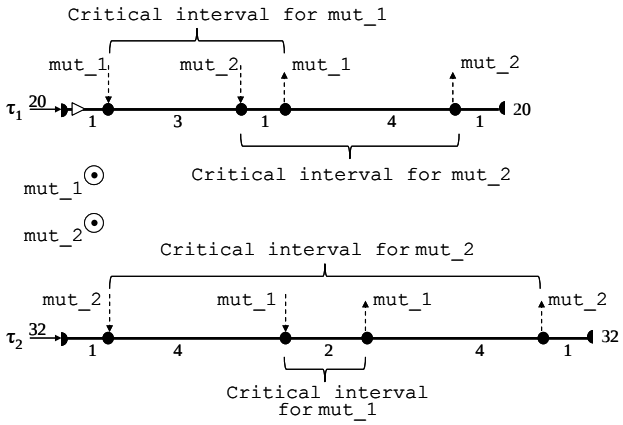

Fig. 2. Application structure represented as a route network

The state of marked-up elements of the route network in Fig. 2 corresponds to the time moment when an instance of the task τ_1 arrived, but has not yet reached the end of the first segment (the cursor is still inside the first segment of the model of the task τ_1) and an instance of the task τ_1 has not arrived yet. Both mutexes are in the state "open".

3. Verification goals and means

Verification methods used in software development assume comparison of formal objects of two types:

- a formal model which adequately reflects software functioning;
- a set of software requirements in the form of assertions which have to be satisfied during its normal functioning.

The goal of verification is to check the consistency between the software model and the specified requirements. Verification means to be used are the formal model of the software, the set of requirements, and the algorithm which checks that the requirements are totally satisfied.

Prospective verification methods of *model checking* are based on representing the system model in the form of finite automata, representing software requirements in form of logical formulae, and performing automated consistency checking of the model and the requirements [5]. These methods are universal in their nature and are applicable to representing a wide range of structural features of systems with sophisticated logical formulae interpreting the requirements to be checked. However, this universality results in high complexity of model and requirements representation, as well as in high effort for implementation of mechanisms for consistency checking. Scalability of these methods also makes a problem, as their complexity grows exponentially with the model size and the number of requirements to be verified.

On the other hand, specialized verification methods used along with universal ones, are aimed at checking particular properties of systems being developed. They usually employ simpler models (e.g., with a reduced inventory of constructive parameters), brief formulations of requirements to be verified, and verification algorithms of low complexity. Feasibility checking of real-time tasks and applications is an example of such specialized methods.

Feasibility ensuring. The most exact method for feasibility checking is the RT-test (Response Time test) [6] based on estimation of the R_i value (response time) for each task τ_i of the given application. If the formula $\forall i \;\; R_i \leq D_i$ holds, then every task and the whole application are feasible.

In case of applications with independent tasks implemented for classical single-core platforms, algorithms for response time estimation provide exact results because they involve analysis of system functioning under the worst (critical) scenarios of system events. In this case, the inequality $R_i \leq D_i$ provides the necessary and sufficient condition for the task τ_i to be feasible.

For independent tasks, the response time R_i equals to the sum $R_i=C_i+I_i$ where C_i is the task τ_i weight factor and I_i is its priority factor. Examples of methods to

estimate I_i may be found in [7] for systems on single-core processors and in [8-10] for multi-core processors.

In case of systems with interdependent tasks, the blocking factor B_i should be added to the response time R_i to reflect an increase of the existence interval of jobs of the type τ_i due to possible existence in the "waiting" state: $R_i=C_i+I_i+B_i$.

In the simplest case, the blocking factor B_i is equal to the maximal weight of those critical intervals, which block the task τ_i [11]. However, the blocking factor may exceed this value, either because some tasks contain intersected critical intervals [12], or because the system is implemented on a multi-core processor [13]. Chained blocking should also be considered, when a particular request for an already locked resource is blocked by several active jobs chained through dependencies of their critical intervals [14]. For systems which allow for chained blocking, complete estimating of task feasibility may be obtained by using special multi-partite oriented graphs, that are presented below.

Ensuring logical correctness. When constructing a multi-task system, the following system properties should be guaranteed:

- integrity of shared information resources;
- absence of options for deadlocks or clinches to occur.

A conventional approach to ensure the integrity of shared information resources is based on mutexes. One should use certain tools to check absences of incorrect usage of the operations `lock/unlock` in the program code.

Presence of critical interval *bundles* in form of nested or intersecting critical intervals (similar to ones in the structure of a two-task application in Fig. 2) is deadlock and clinch prone. A *deadlock* is such a system state, in which all tasks are trapped in the "waiting" state and the system hangs as there are no tasks in the "running" state; i.e., there is no activity capable to perform the `unlock` operation which corresponds to the requested information resource. A *clinch* is such a system state, where a number of tasks are bundled by a ring of mutual blockings; however, there still is a subset of tasks capable to be executed indefinitely long.

One of the possible ways to check a model of a multitask system for potential deadlocks or clinches is to construct an oriented state graph, whose vertices represent reachable states and arcs represent possible transitions from one state into another. A deadlock is possible if the graph contains a vertex with no outgoing arcs. Checking the state graph for potential clinches is more complicated.

For the model in Fig. 2 the state graph may be constructed and analyzed manually – it contains only several dozen vertices and arcs. However, the number of vertices and arcs in such a graph grows exponentially with the number of tasks. In RTS software applications used in practice, there are dozens of tasks and shared information resources – for such systems construction and analysis of the state graph is unfeasible not only manually, but with much more powerful tools as well.

A conventional way to ensure logical correctness of an application implementation with interdependent tasks is based on providing the synchronizing mechanisms of the mutex type with special protocols for access to shared information resources [15]. The Priority Inheritance Protocol (PIP) ensures that no *priority inversion* occurs; i.e., no blocking of higher priority jobs takes place due to

excessive activation of medium priority tasks. The PIP protocol acts as follows. If a resource turns out to be locked by a job $\tau_j^{(y)}$ with a lower priority than the job $\tau_i^{(x)}$ which executes the operation `lock` on a mutex (i.e., at the moment of issuing a request for this resource), then temporally, until the resource becomes available, the priority of $\tau_j^{(y)}$ is increased up to the priority of $\tau_i^{(x)}$ (i.e., $\tau_j^{(y)}$ temporally inherits the priority of $\tau_i^{(x)}$). The Priority Ceiling Protocol (PCP) prevents not only priority inversion, but mutual blockings as well.

4. Bundle dependency graph

A multi-partite oriented graph (called *bundle dependency graph*) is aimed at checking logical correctness of a multi-task application model. The construction method and analysis method of the bundle dependency graph are illustrated with a model of an application consisting of fowr interdependent tasks (see Fig. 3 and Fig. 4).

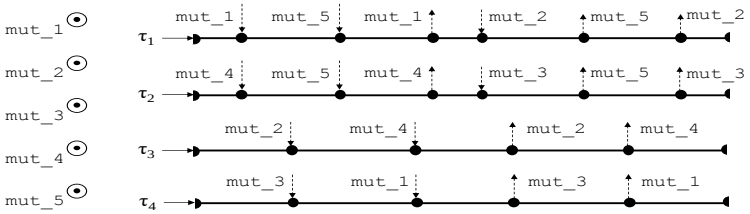

Fig. 3. The initial system state which allows for clinches

The initial system state in Fig. 3 is presented with the means of route networks: all mutexes are in the state “open” (e.g., all shared resources are available) and no task is activated. One of reachable states of this application is depicted in Fig. 4:

- mutex `mut_1` is open, other mutexes are locked;
- an instance of the task τ_1 is waiting for the mutex `mut_2` to open (the critical interval associated with this mutex is occupied by an instance of the task τ_3);
- an instance of the task τ_2 is waiting for the mutex `mut_5` to open (the critical interval associated with this mutex is occupied by an instance of the task τ_1);
- an instance of the task τ_3 is waiting for the mutex `mut_4` to open (the critical interval associated with this mutex is occupied by an instance of the task τ_2);
- an instance of the task τ_4 is in the “running” state and executes the critical interval guarded by the mutex `mut_3`.

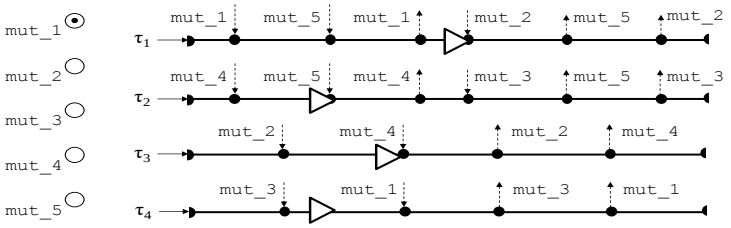

Fig. 4. A ring of mutual waiting: $\tau_1 \rightarrow \tau_3 \rightarrow \tau_2 \rightarrow \tau_1$

Thus, in the state depictured in Fig. 4 jobs of the types τ_1, τ_2, τ_3 are linked in a ring of mutual waits. This state is not a deadlock because an instance of the task τ_4 may continue its execution and then other jobs of the type τ_4 may be generated and executed an arbitrary number of times, performing operations `lock` and `unlock` on mutexes `mut_1` and `mut_3`.

Further analysis shows that the system state in Fig. 4 is not the only reachable state of mutual blocking for the modeled system. To demonstrate this, let's construct the bundle dependency graph for this application.

Let's call two critical intervals of the task τ with respect to the resources g and g^* *bundled* (forming a *bundle* $L=\langle\tau, g, g^*\rangle$), if they intersect; i.e., they contain common segments of the task code. Each bundle $L=\langle\tau, g, g^*\rangle$ consists of three sectors. In the starting (*head*) sector the task τ has access to the *head resource* g of the bundle. At the central sector the task τ has access to both resources – the head resource g and *additional resource* g^*. At the ending segment one of the bundle resources is released by the task τ. The central segment consists of segments which form an intersection of bundled critical intervals.

The task τ_1 contains two bundles: $L_a=\langle\tau_1, g_1, g_5\rangle$ and $L_b=\langle\tau_1, g_5, g_2\rangle$ (Fig. 3). The segment between operators `lock(mut_5)` and `unlock(mut_1)` in the bundle L_a is a common sector of critical intervals with respect to the head resource g_1 and the additional resource g_5. The segment between operators `lock(mut_2)` and `unlock(mut_5)` in the bundle L_b is an intersection of bundled critical intervals. The task τ_2 contains two bundles: $L_c=\langle\tau_2, g_4, g_5\rangle$ and $L_d=\langle\tau_2, g_5, g_3\rangle$. The task τ_3 contains one bundle $L_e=\langle\tau_3, g_2, g_4\rangle$, and the task τ_4 contains yet another bundle $L_e=\langle\tau_4, g_3, g_1\rangle$.

A necessary condition for rings of mutual waits to appear is the presence in the software application of such bundle pairs, for which the following dependency relation holds. A bundle $L_x=\langle\tau_i, g_a, g_b\rangle$ *depends* on the bundle $L_y=\langle\tau_j, g_c, g_d\rangle$, if τ_i and τ_j are different tasks and $g_b \equiv g_c$. In other words, bundle L_x depends on bundle L_y, if L_x and L_y belong to different tasks and the head resource of the bundle L_y coincides with the additional resource of the bundle L_x.

The fact of dependency of the bundle L_x on the bundle L_y is denoted by the symbol "$\rightarrow$" ($L_x \rightarrow L_y$). With these notations the following dependencies hold: $L_a \rightarrow L_d$, $L_b \rightarrow L_e$, $L_c \rightarrow L_b$, $L_d \rightarrow L_f$, $L_e \rightarrow L_c$, $L_f \rightarrow L_a$ for the model in Fig. 3. These dependencies may be represented graphically by constructing a multi-partite oriented *bundle dependency graph*: each bundle is represented by a graph vertex; an arc from vertex L_x to vertex L_y means that the bundle L_y depends on the bundle L_y. Each task corresponds to a partite in the constructed graph.

A bundle dependency graph for a system represented with the model in Fig. 3 is depicted in Fig. 5.

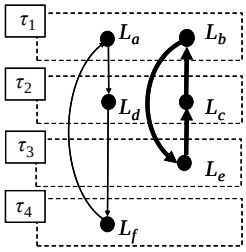

Fig. 5. Bundle dependency graph

5. Inter-partite routes and contours

Identifying inter-partite routes and contours in the bundle graph is important for static analysis of systems with interdependent tasks. A route in an oriented graph is a sequence of its arcs such that:

1. Two adjacent arcs (the preceding and the following) have a common vertex; the preceding arc is an incoming arc of the common vertex and the following arc is its out-going arc.

2. Each vertex of the route occurs in it only once (the route has no self-crossings and its starting and ending vertices are different).

In a multi-partite graph a route is an *inter-partite route*, if no two vertices of this route belong to the same graph partite.

A contour of an oriented graph is a closed sequence of adjacent arcs without self-crossings (the first arc is an out-going arc for the same vertex, which the last arc is an incoming arc for).

A contour of a multi-partite graph is an *inter-partite contour*, if no two vertices of this contour belong to the same graph partite.

Analysis of the bundle dependency graph structure is important due to the fact that rings of mutual waits in a multi-task system become possible, if and only if this graph contains inter-partite contours.

A bundle dependency graph in Fig. 5 has two inter-partite contours. Each one represents an achievable marking-up which characteristic feature is that some tasks are mutually blocked. In particular, the inter-partite contour $L_b \rightarrow L_e \rightarrow L_c \rightarrow L_b$ (distinguished with thick lines in Fig. 5) corresponds to a ring of mutual waits, which bounds the tasks τ_1, τ_3, and τ_2 in the system state represented in Fig. 4. The bundle dependency graph in Fig. 5 contains another inter-partite contour – that of $L_a \rightarrow L_c \rightarrow L_e \rightarrow L_a$. This contour corresponds to a potential variant of event sequence which results in a ring of mutual waits among the tasks τ_1, τ_2, and τ_4.

The practice of constructing multi-task software applications demonstrate that the number of critical intervals in each particular task does not exceed some fixed upper bound when the number of tasks grows. Therefore, the upper bound of the number of vertices in the bundle dependency graph grows linearly with the number of tasks. It follows that construction and analysis of the bundle dependency graph for real-time multi-task applications may be done within reasonable time.

If there are no inter-partite contours in a bundle dependency graph, then the respective application is free of deadlocks and clinches not only with the PCP

protocol, but with PIP and PCP protocols as well. It was demonstrated in [13] that the blocking factor may turn out to be less with PIP than with PCP. This is valid for implementations of systems on both single- and multi-core processor platforms. Therefore, applying the technique of bundle dependency graphs to static analysis of RTS applications may increase the efficiency of processor resources usage.

6. Graph of bundles and critical intervals

It was demonstrated in [14] that with the PIP protocol chained blocking may occur on both single- and multi-core processors when a particular request for an already locked resource may be blocked by several active jobs chained by dependencies of their critical intervals. To estimate the blocking factor in such situations, a variety of multi-partite graphs – a *bundle and critical interval graph* – may be used.

Fig. 6a presents the structure of a system composed of three tasks and two resources. Studying the chart in Fig. 6b demonstrates that a request for the resource g_1 by the task τ_1 results in its blocking, first, directly by the task τ_3 which owns the required resource and then (indirectly, at the time t=14) by the task τ_2 which owns a resource needed not by the task τ_1 itself, but rather by the task τ_3 which blocks the task τ_1. Thus, critical intervals of the tasks τ_3 and τ_2 form two links of a chain which blocks execution of the task τ_1.

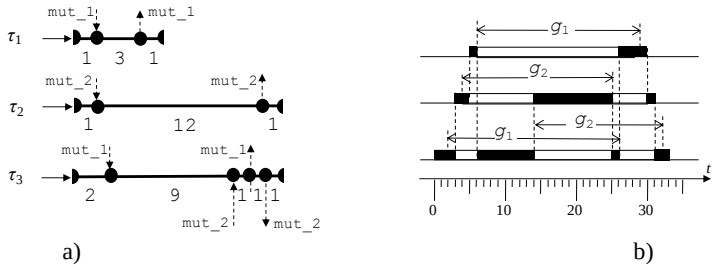

Fig. 6. Chained blocking

The following question comes naturally: “How many active tasks may be involved in a blocking chain?” This may be answered through analysis of the *bundle and critical interval graph*, introduced below. Its distinguishing feature is that each graph partite which corresponds to a particular task τ_i, consists of vertices corresponding to:

- bundles of critical intervals of the task τ_i code;
- critical intervals within the bundles of the task τ_i code;
- free critical intervals (critical intervals within the task τ_i code which belong to no bundle).

The arcs of the bundle and critical interval graph are constructed according to the following rules.

Rule 1. Two vertices L_a and L_b (corresponding to the bundles L_a and L_b), are connected by an arc from L_a to L_b, if L_a and L_b belong to different graph partites and the head resource of the bundle L_b coincides with the additional resource of the bundle L_a.

Rule 2. An arc is drawn from vertex $G(g)$ to vertex L, if these vertices belong to different graph partites and the head resource of L coincides with the resource g.

Rule 3. An arc is drawn from vertex L to vertex $G(g)$, if the resource g coincides with the additional resource of bundle L.

Fig. 7 depicts a bundle and critical interval graph for a software application configuration of Fig. 6. Let's add a parameter W – the arc weight – to each arc of this graph. The weight of an arc incoming into the vertex $G(g)$ equals to the computational effort needed to execute the critical interval of the vertex $G(g)$. The weight of an arc incoming into the vertex L equals to the computational effort needed to execute the heading critical interval of the bundle corresponding to the vertex L.

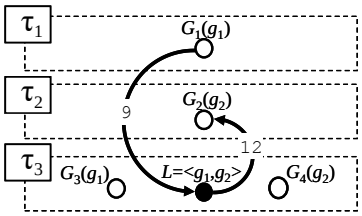

Fig. 7. A bundle and critical interval graph

To find-out what blocking chains are feasible at entering the k-th critical interval of the task τ_i, blocking routes should be constructed for the vertex G which corresponds to this critical interval. The route weight equals to the sum of the weights of its composing arcs. Chained blocking is feasible, if the blocking route contains more than one arc.

The route $\langle G_1, L, G_2\rangle$ of the graph in Fig. 7 contains two arcs with the total weight $W(G_1, L)+W(L, G_2)=9+12=21$. Therefore, at entering a critical interval associated with the resource g_1, chained blocking for the task τ_1 is feasible, with its maximal duration up to 22 time units.

A scenario of system events involving the occurrence of a chained blocking, which corresponds to the blocking route $\langle G_1, L, G_2\rangle$, is depicted in Fig. 6b.

7. Conclusion

When developing a RTS, verification issues referred to logical and dynamic correctness of the system as a whole should be addressed along with similar issues referring to individual system tasks. Establishing logical correctness assumes checking that erroneous situations, such as infinite waits for signal messages (deadlocks and clinches) among interdependent tasks, are impossible. Static verification of relevant evidences may be done through construction and analysis of a multi-partite bundle dependency graph of the system under consideration.

The necessity to ensure efficient usage of computing resources with a warranty of timely execution of all software application tasks is among the RTS issues of its dynamic correctness. Possible duration of high-priority tasks blocking because of their waiting for locked information resources to be released by low-priority tasks should be taken into account. To estimate the blocking duration in systems with

chained blocking of tasks, application of the described technique of multi-partite oriented bundle and critical interval graph may be a reasonable solution.

The proposed method allows for unlimited scalability with respect to the number of tasks and shared resources of the application under consideration because of the linear complexity of the respective algorithm for construction and analysis of the respective bundle and critical interval graph derived from the application structure. As by now, the method has been tried only on simple artificial examples and is still to be validated in practical engineering with real-time multi-task applications in order to be further developed into a specialized verification tool to check feasibility of applications, which structure may be formally represented in the described way. Its special feature is that with this method the application control environment may be timely informed of an occurring deadlock or clinch in the application, which allows it to properly react to such run-time exception.

***Acknowledgments*:** This work was partially financially supported by the Government of the Russian Federation, Grant 074-U01.

References

1. Davidenko, K. Y. Software Engineering for Automatic Control Systems of Technological Processes. Design of Real-Time Systems, Parallel, and Distributed Applications. Moscow, Energoatomizdat, 1985. 183 p. (in Russian).
2. Liu, C., J. Layland. Scheduling Algorithms for Multiprocessing in a Hard Real-Time Environment. – Journal of the ACM, Vol. **20**, 1973, No 1, pp. 46-61.
3. Nikiorov, V. V. Feasibility of Real-Time Applications on Multi-Core Processors. – SPIIRAS Proceedings, Issue 8, Nauka, St. Petersburg, 2009, pp. 255-284 (in Russian).
4. Nikiforov, V. V., V. A. Pavlov. Structured Models for Multi-Task Software System Analysis. – Information-Measuring and Control Systems, No 9, 2011, pp.19-29 (in Russian).
5. Karpov, Y. Model Checking. St. Petersburg, BHV-Petersburg, 2010. 560 p. (in Russian).
6. Bini, E., G. C. Buttazzo, G. M. Buttazzo. Rate Monotonic Analysis: The Hyperbolic Bound – IEEE Transactions on Computers, Vol. **52**, July 2003, No 7, pp. 933-942.
7. Laplante, P. A. Real-Time Systems Design and Analysis. John Wiley & Sons, Inc., 2004. 530 p.
8. Baker, T. Multiprocessors EDF and Deadline Monotonic Schedulability Analysis. – In: Proc. of 24th IEEE Real-Time Systems Symposium, 2003, pp. 120-129.
9. Andersson, B., S. Baruah, J. Jonsson. Static-Priority Scheduling on Multiprocessors. – In: Proc. of 22nd IEEE Real-Time Systems Symposium, London, 2001, pp. 193-202.
10. Andersson, B. Global Static-Priority Preemptive Multiprocessor Scheduling with Utilization Bound 38%. – In: Proc. of 12th International Conference on Principles of Distributed Systems, Egypt, Luxor, December, 2008, pp. 73-88.
11. Liu, J. W. S. Real-Time Systems. NJ, Prentice Hall, 2000. 590 p.
12. Nikiforov, V. V., V. I. Shkirtil. Route Networks – A Graphical Formalism for Representing the Structure of Real-Time Software Applications. – SPIIRAS Proceedings, Issue 14, SPb: Nauka, 2010, pp. 7-28 (in Russian).
13. Nikiforov, V. V., V. I. Shkirtil. Estimating the Task Blocking Factor in Real-Time Systems with Multi-Core Processors. – SPIIRAS Proceedings, Issue 4(27), SPb: Nauka, 2013, pp. 93-106 (in Russian).
14. Nikiforov, V. V., V. I. Shkirtil. Chained Task Blocking in Real-Time Systems. – Information Measuring and Control Systems, 2013, No 7, pp. 17-21 (in Russian).
15. Sha, L., R. Rajkumar, J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. – IEEE Transactions on Computers, Vol. **39**, September1990, No 9, pp. 1175-1185.