Multitasking in Embedded System Designs

https://doi.org/10.3991/ijoe.v15i09.10631

Radoslav Mavrevski (✉), Metodi Traykov,
South-West University, Blagoevgrad, Bulgaria
radoslav_sm@abv.bg

Ivan Trenchev
South-West University, Blagoevgrad, Bulgaria
University of Library Studies and Information Technologies Sofia, Bulgaria

Abstract—It is common knowledge in Information Technology (IT) that an embedded system is based on microprocessor and is built to control a function or a range of functions. Although, it is not designed to be programmed by the end user in the same way that a PC is, it is designed to perform one particular task with choices and different options [1-5]. Multitasking is a method by which multiple tasks, also known as processes, share common processing resources, such as CPU. The main aim of this paper is analysis of the design of the embedded systems and a focus on mid-level abstractions for concurrent programs.

Keywords—Embedded System, multitasking, design

1 Introduction

The main "parts" that consist an embedded system are: Processor, Memory, Peripherals, Software and Algorithms.

The main criteria for the **processor** are whether it can provide the processing power needed to execute the tasks within the system. Sometimes it occurs that the tasks are either underestimated in terms of their size and/or complexity or that the specification is beyond the processor's capabilities. Usually, these issues are in bigger scale because of the performance measurement used to judge the processor. They may execute completely out of cache memory and thus give an artificially high performance level which the final system cannot meet because its software does not fit in the cache. The software overheads for high level languages, operating systems and interrupts may be higher than expected. These are all issues that can turn a paper design into failed reality [2-4]. While processor performance is important and consists the first gating criterion, there are others such as cost, power consumption, software tools and component availability.

Memory is an important part of any embedded system design, especially nowadays that the requirements and the data are huge. The software influences the resources of a computer system and its memory. Memory can determine how the soft-

ware is designed, written and developed. It primarily performs two functions within an embedded system: 1) It provides storage for the software that it will run; and 2) It provides storage for data such as program variables and intermediate results, status information and any other data that might be created throughout the operation

It is of high importance that an embedded system must be able to communicate with the outer world via **peripherals**. Input peripherals are usually associated with sensors that measure the external environment and thus effectively control the output operations that the embedded system performs [2-4]. So, an embedded system can be designed on a three-stage pipeline. Data and information are being put into the first stage of the pipeline; the second stage carries out the processes and the third stage outputs results and data.

There are five main types of peripherals: Binary outputs, which are simple external pins whose logic state can be controlled by the processor to either zero (off) or one (on); Serial outputs, which are interfaces that send or receive data using one or two pins in a serial mode; Analogue values, which are interfaces between the system and the external environment needed to be converted from analogue to digital and vice versa; Displays, which can vary from simple LEDs and seven segment displays to small LCD panels; Time derived outputs (timers and counters).

The **software** components within an embedded system often encompass the technology that adds value to the system and defines what it does and how well it does it [1-9]. The software includes several different components: Customization and configuration; Applications (Modules); Operating system; Error handling and Maintenance.

Algorithms are the key components of the software that makes an embedded system behave in the way that it does, so as to fulfill its purpose. They can vary from mathematical processing to models of the outer environment which are used to interpret and take advantage of information that derives from the external sensors and thus generate control signals. Nowadays with the digital technology such, the algorithms that digitally encode the analogue data are defined by standard bodies. Getting the right implementation is very important since, for example, it may allow the same function to be executed on cheaper hardware (Efficiency). As most embedded systems are designed to be commercially successful, the selection of the "right" algorithm is extremely important.

2 Material and Method

Before we move further on, to the part of multitasking, it is necessary to explain what an operation system is. It is a software environment that provides a buffer between the user and the low-level interfaces to the hardware within a system. It provides a constant interface and a set of utilities to enable users to utilize the system quickly and efficiently [7]. Furthermore, software is allowed to be moved from one system to another and thus can make application programs hardware independent. Very often there are program debugging tools included. Therefore, the testing process is completed in lesser time.

Most of the embedded systems nowadays demand a multitasking operating system. It is an operating system that can run multiple applications simultaneously and provide intertask communication and control. A multitasking operating system works by dividing the processor's time into discrete time slots. Each application or task demands a specific number of time slots to complete its execution. The operating system kernel decides which task can have the next slot, so instead of a task executing continuously until completion, its execution is interleaved with other tasks. This sharing of processor time between tasks gives the illusion to each user that he is the only one using the system [8].

Further on, comes the analysis of the mechanisms that are used in software in order to offer concurrent execution of sequential code. With this, multitasking can be achieved. There are multiple reasons for executing more than one sequential program concurrently and they all involve timing. One of the main reasons is to improve responsiveness by avoiding situations where long-running programs can block a program that responds to external stimuli, such as sensor data or a user request [1-3]. Faster responsiveness reduces latency, which is in fact, the time between the occurrence of a stimulus and the response. Moreover, another reason is to improve performance by allowing a program to be executed simultaneously on multiple cores or processors. Furthermore, one other reason is to directly control the order of external actions. A program may need to perform some action, such as updating a display or saving data that was recently added/modified, at particular times, no matter what other tasks might be executing at the same time.

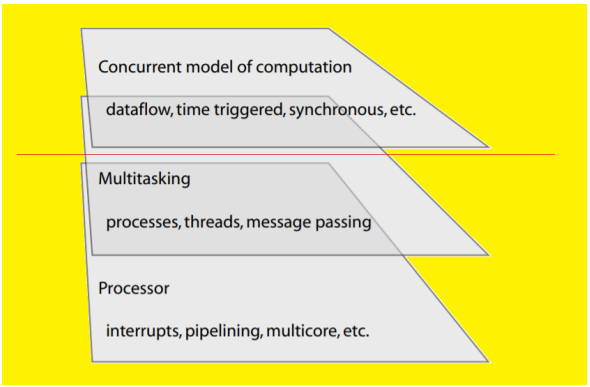

Fig. 1. Layers of abstraction for concurrency in programs

Figure 1 shows that multitasking are in fact mid-level techniques which are directly related with the low-level and high-level mechanisms. Embedded system designers usually use these mid-level mechanisms to build applications, but it is becoming increasingly common for designers to use the high-level mechanisms instead. The designer builds a model of computation using a software tool. This model is then automatically or semi-automatically translated into a program that exploits the mid-level or low-level mechanisms. The process of translation is widely known as code generation or auto coding.

The mechanisms are provided by an operating system, a microkernel, or a library of procedures. They can be rather difficult to implement correctly, and because of that, the work should be done by very experienced people (experts).

3 Results and Discussion

When it comes to the programming language that application programmers use to develop software, a language that expresses a computation as a sequence of operations is called an imperative language. C is an example of imperative language. In Figure 2 there is an example of a program written in C which implements a commonly used design patter which is known as the observer pattern [10].

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  int x;                              // Value thatgets updated.
4  typedef void notifyProcedure(int);     // Typeof notify proc.
5  struct  element {
6      notifyProcedure* listener;      // Pointer to notify procedure.
7      struct element* next;             //  Pointer to the next item.
8  };
9  typedef struct element element_t;    // Type of  list elements.
10  element_t*  head =  0;               // Pointer to start  of list.
11  element_t* tail =  0;                //  Pointer to end of list.
12
13   // Procedure to  add a listener.
14   void addListener(notifyProcedure*  listener) {
15      if (head ==  0)  {
16          head =  malloc(sizeof(element_t));
17          head->listener =  listener;
18          head->next =  0;
19          tail =  head;
20      }  else  {
21          tail->next =  malloc(sizeof(element_t));
22          tail =  tail->next;
23          tail->listener =  listener;
24          tail->next =  0;
25      }
26   }
27   // Procedure to  update x.
28   void update(int  newx) {
29      x  =  newx;
30      II Notify listeners.
31      element_t* element =  head;
32      while (element  != 0)  {
33          (*(element->listener))(newx);
34          element =  element->next;
35      }
36   }
37   // Example  of notify  procedure.
38   void print(int  arg) {
39      printf("%d ", arg);
40   }
```

Fig. 2. An example of a program written in C (observer pattern)

In the following example, an update procedure changes the value of a variable x. The other programs or other parts of the program, which are called “observers”, will be notified every time x is changed by calling a callback procedure. For instance, the value of x might be displayed by an observer on a screen. Whenever the value chang-

es, the "observer" needs to be notified so that it can update the display on the screen. The example below demonstrates a main procedure that uses the procedures defined in Figure 2:

```
1     int main(void)  {
2             addListener(&print);
3             addListener(&print);
4             update(1);
5             addListener(&print);
6             update(2);
7             return 0;
8     }
```

In this example, the program registers the *print* procedure as a callback twice, then performs an update, changing the value of x (x = 1), then registers the *print* procedure again, and finally does another update, changing the value of x (x = 2). The *print* procedure prints the current value of the variable x, so the final output when executing this program will be 1 1 2 2 2.

In general, a program in C specifies an order of steps, where each step changes the state of the memory in the machine. In C, the state of the memory is represented by the values of variables [1-4].

In the example which was described in Figure 2, the state of the memory has the value of variable x which is a global variable. A global variable is visible through all the procedures and "parts" of the program. Furthermore, there is a list of elements pointed to by the variable head, which consists another global variable. The list itself is represented as a linked list, where each element in the list contains a function pointer referring to a procedure to be called when x changes [1-4].

In computer science, a linked list is a linear collection of data elements, called nodes, each pointing to the next node by means of a pointer. It is a data structure consisting of a group of node which together represents a sequence. Under the simplest form, each node is composed of data and a reference (in other words, a link) to the next node in the sequence. This structure allows for efficient insertion or removal of elements from any position in the sequence during iteration. More complex variants add additional links, allowing efficient insertion or removal from arbitrary element references. Below, there is a definition of a linked list for the program in Figure 2:

```
1 typedef void notifyProcedure(int);
2 struct  element {
3   notifyProcedure* listener;
4   struct  element* next;
5 };
6 typedef struct element element_t;
7 element_t* head =0;
8 element_t* tail =0;
```

These are all examples of code that runs sequentially, line after line. Complex programs can create difficulties handing values and the state of memory. The problems get much worse when concurrency gets into the mix. C programs with mid-level concurrency mechanisms such as threads are extremely difficult and prone to errors.

Threads are imperative programs that run concurrently and share a memory space. One of the advantages is the fact that the programs can access each other's variables. The term "threads" is widely used to refer to particular ways of constructing programs that share memory. In this paper however, the term will refer to any mechanism where imperative programs run concurrently and share memory.

Most operating systems provide a higher-level mechanism which is provided in the form of a series of procedures that a programmer can exploit. Such procedures typically conform to a standardized API (application program interface), which makes it possible to write programs that are portable, which in fact is very useful as they run on multiple processors and/or multiple operating systems. Pthreads (or POSIX threads) is such an API. It is integrated into many modern operating systems. Pthreads defines a set of C programming language types, functions and constants [1-4]. In Figure 3, there is an example of a multithreaded program.

```
1   #include <pthread.h>
2   #include <stdio.h>
3   void* printN(void* arg) {
4           int i;
5           for (i = 0; i < 10; i++) {
6                   printf("My ID: %d\n", *(int*)arg);
7           }
8           return NULL;
9   }
10   int main(void) {
11          pthread_t threadID1, threadID2;
12          void* exitStatus;
13          int x1 = 1, x2 = 2;
14          pthread_create(&threadID1, NULL, printN, &x1);
15          pthread_create(&threadID2, NULL, printN, &x2);
16          printf("Started threads.\n");
17          pthread_join(threadID1, &exitStatus);
18          pthread_join(threadID2, &exitStatus);
19          return 0;
20  }
```

Fig. 3. Multithreaded program with Pthreads

The *printN* procedure, the procedure that the thread begins executing, is known as the start routine. It prints the argument passed to it 10 times and then exits, which will force the thread to end. The *main* procedure creates two threads; each of one will execute the start routine. The first one, which is created on line 14, will print the value 1. The second one, which is created on line 15, will print the value 2. During the execution of the program, values 1 and 2 will be the outputs that will be displayed on screen, in some mixed order that depends on scheduler of the thread. Each time the program runs, the results will be different orders of 1's and 2's.

In addition, the *pthread_create* procedure creates a thread and returns immediately. The start routine may or may not have actually started running when it returns. Lines

17 and 18 use pthread join to ensure that the main program does not terminate before the threads have finished. Without these two lines, running the program may not yield any output at all from the threads [1-5]. Generally, a start routine may return or not. In embedded applications, it is common to define start routines that never return**.** For instance, the start routine might run forever and produce an output display periodically. If the start routine does not return, then any other thread that calls its pthread join will be blocked indefinitely.

The center part of an implementation of threads is a scheduler that chooses which thread will be the one executed next when a processor is available to execute one of them. The choice may be based on fairness, where the principle is to give every active thread an equal opportunity and time to run, on timing limits, or in terms of how important it is according to its priority. The first matter that concerns the programmers is how and when the scheduler is invoked. A simple technique known as cooperative multitasking does not interrupt a thread unless the thread itself calls a specific procedure or one out of a set of procedures. For instance, the scheduler may intervene each time an operating system service is invoked by the currently executing thread. Respectively, an operating system service is invoked by calling a library procedure. Each thread has its own stack, and when the procedure call is made, the return address will be pushed onto the stack. If the scheduler determines that the currently executing thread should continue to execute, then the requested service is completed and the procedure returns as normal. On the other hand, if the scheduler determines that the thread should be interrupted and another thread should be selected for execution, then instead of returning, the scheduler makes a record of the stack pointer of the currently executing thread, and then changes the stack pointer to point to the stack of the selected thread. Moreover, it returns as normal by popping the return address off the stack and resuming execution, but this time in a new thread.

Concerning the main drawback of cooperative multitasking, it is that a program may run for a long time without making any operating system service calls, in which case other threads will be put to hold almost indefinitely. To avoid this, most of the operating systems out there, offer an interrupt service routine that runs at certain time intervals. This routine maintains a system clock, which provides application programmers with a solution to obtain the current time of day and enables periodic invocation of the scheduler with the use a timer interrupt.

In multitasking, two concurrent pieces of code race to access the same resource, and the exact order in which their accesses occur affects the results of the program. Not all race conditions are as bad as the previous example, where some outcomes of the race cause catastrophic failure. One effective solution to avoid these disasters is by taking advantage of a mutual exclusion lock (or mutex). The following piece of code is such an example:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void addListener(notifyProcedure*  listener) {
  pthread_mutex_lock(&lock);
  if (head == 0)  {
```

```
      ...
    }  else  {
      ...
    }
    pthread_mutex_unlock(&lock);
  }
```

On the first line a global variable called *lock* is created and initialized. Inside the *addListener* procedure acquires the lock. The principle is that only one thread can hold the lock each time. The *pthread_mutex_lock* procedure will block until the calling thread can acquire the lock. When the procedure *addListener* is called by a thread and begins executing, pthread mutex lock does not return until no other thread holds the lock. Once it returns, this calling thread holds the lock. The *pthread_mutex_unlock* call at the end releases the lock. It is a serious error in multithreaded programming to fail to release a lock.

A mutual exclusion lock prevents two threads from simultaneously accessing or changing a shared resource of the memory. The code between the lock and unlock is a critical section. At any cost, only one thread can be executing code in such a critical section. A programmer may need to ensure that all accesses to a shared resource are similarly protected by locks [1-4].

However, the mutex added in the previous example is not enough to avoid this problem. The mutex does not prevent thread A from being halted. So, there is need to protect all accesses of the data structure with mutexes, which can be done with altering *update* procedure as described below:

```
  void update(int  newx) {
    x = newx;
// Notify listeners.
pthread_mutex_lock(&lock);
element_t* element = head;
while (element != 0) {
      (*(element->listener))(newx);
      element = element->next;
    }
    pthread_mutex_unlock(&lock);
  }
```

The above code will refrain the update procedure from reading the list data structure while it is being altered by any other thread.

As mutex can be used frequently in programs, the risk of deadlock gets bigger. A deadlock occurs when some threads become permanently halted struggling to acquire locks. As an example, if thread A holds lockl and then stops trying to acquire lock2, which is taken by thread B, and then thread B stops trying to get lockl. This will lead the system to crash and the program needs to be aborted.

Deadlock can be rather difficult to avoid. One simple technique is to use only one lock throughout an entire multithreaded program. On the other hand, that technique

does not launch very modular programming. In addition, it can make it difficult to meet real-time constraints because some shared resources (e.g., displays) may need to be held long enough to cause deadlines to be missed in other threads [1-3].

Sometimes in a very simple microkernel, it is easy to use the enabling and disabling of interrupts as a single global mutex. If there is a single processor and that interrupts are the only available mechanism by which a thread may be halted, then disabling interrupts avoids the suspension of a thread. On the other hand, in most Operating Systems, threads can be halted for various reasons, thus this technique won't give back.

Moving further, another technique is to ensure that when there are more than one mutex locks, each thread acquires the locks in the same order. This can be difficult to guarantee, however, for several reasons:

- Most programs are written by multiple people, and the locks acquired within a procedure are not part of the signature of the procedure. Thus, this technique relies on very careful and constant cooperation and writing down throughout a programming team. Everytime a lock is added, and then all parts of the program that acquire locks may have to be changed, to sustain its "vitality".
- Correction of the source code may be proved to be extremely difficult. If a programmer wants to call a procedure that acquires lockl, then it must first release any locks it holds. As soon as it releases those locks, it may be halted, and the resource that it held those locks to protect may be modified. Once it has acquired lock1, it must then reacquire those locks, but it will then need to assume it no longer knows anything about the state of the resources, and it may have to redo considerable work.

Of course, there are many other ways to avoid deadlock. For example, a particularly elegant technique synthesizes constraints on a scheduler to prevent deadlock [9]. However, most of them either inflict serious constraints and problems on the programmer or demand considerable sophistication to apply, which suggests that the problem may be with the concurrent programming model of threads [1-5].

Threads also suffer from problems that have to do with the memory model of the programs. Implementation of threads gives some kind of memory consistency model, which defines how variables that are read and written by various threads appear to them. Reading a variable should return the last value written to the variable. For instance, in a program, all variables are initialized with value 0 (zero), and thread A executes the following two lines:

```
1   x = 5;
2   w = y;
```

while thread B executes the following two lines:

```
1   y = 5;
2   z = x;
```

Then, after both threads have executed these lines, the expected result is that at least one of the two variables w and z will have value 5 (five). This is known as sequential consistency [10]. Sequential consistency means that the result of any execution is the same as if the operations of all threads are executed in some sequential

order, and the operations of each individual thread appear in this sequence in the order specified by the thread.

On the other hand, sequential consistency is not working everytime by most implementations of Pthreads. In fact, such a guarantee is rather difficult to be achieved on modern processors and modern compilers. A compiler, for instance, is free to reorder the instructions in each of these threads because there is no dependency between them. There is also a chance that the hardware might reorder them instead. A solution to this is to very carefully guard such accesses to shared variables using mutual exclusion locks.

Multithreaded programs can be very difficult to understand. Furthermore, it can be difficult to trust the programs because problems in the code may not show up in the testing process. A program may have the possibility of deadlock, for example, but nonetheless run correctly for years without the deadlock ever appearing [1-5].

```
void update(int  newx) {
    x =  newx;
    // Copy   the list
    pthread_mutex_lock(&lock);
    element_t*  headc =  NULL;
    element_t* tailc = NULL;
    element_t* element = head;
    while (element != 0)  {
        if (headc == NULL)  {
              headc = malloc(sizeof(element_t));
              headc->listener = head->listener;
              headc->next = 0;
              tailc = headc;
        } else {
              tailc->next = malloc(sizeof(element_t));
              tailc = tailc->next;
              tailc->listener = element->listener;
              tailc->next = 0;
        }
        element = element->next;
     }
     pthread_mutex_unlock(&lock);

    // Notify listeners using the copy
    element = headc;
    while (element != 0)  {
        (*(element->listener))(newx);
        element = element->next;
    }
}
```

Fig. 4. Modified update procedure

In Figure 4, there is a modified version of the *update* procedure that was described earlier. This code does not hold lock when it calls the listener procedure. Instead, it holds the lock while it constructs a copy of the list of the listeners, and then it releases the lock. After releasing the lock, it uses the copy of the list of listeners to notify the listeners. However, it carries a potentially critical problem that is hard, if not completely impossible, to be detected in testing. For example, thread A calls *update* with newx = 0, indicating "All systems good.". A gets suspended just after releasing the

lock, but before performing the notifications. While it is suspended, thread B calls *update* with newx = 1, indicating "Emergency! The engine is off!" Suppose that this call to update completes before thread A gets a chance to resume. When thread A resumes, it will notify all the listeners, but it will notify them of the wrong value! If one of the listeners is updating a pilot display for an aircraft, the display will indicate that all systems are normal, when in fact the engine is on fire [1-3]. This is an example of a very serious issue that could be fatal (loss of human lives) and thus it must be solved effectively.

Last but not least, processes are also important to keep an eye on when it comes to embedded systems. Processes are imperative programs with their own memory spaces. One of their main characteristics is that they cannot refer to each other's variables, and consequently they do not demonstrate the same difficulties as threads. Communication between the programs must occur through the use of mechanisms provided by the operating system, a library, or microkernel.

When it comes to the implementation, processes generally demand hardware support in the form of a memory management unit or MMD. The MMD protects the memory of one process from accidental, not desired reads or writes by another process and also provides address translation, giving each process the "illusion" of a fixed memory address space that is the same for all processes. When a process accesses a memory location in that address space, the MMD shifts the address to refer to a location in the portion of physical memory allocated to that process [1-5].

Operating systems offer various mechanisms, often even including the ability to create shared memory spaces, which of course opens the programmer to all the potential difficulties of multithreaded programming. After all, in order to achieve concurrency, the processes must able to communicate. A flexible mechanism for communicating between processes is message passing. One process creates a chunk of data, deposits it in a carefully controlled section of memory that is shared, and then notifies other processes that the message is ready. Those other processes can block waiting for the data to become ready. Message passing requires some memory to be shared, but it is implemented in libraries that are presumably written by experts. An application programmer invokes a library procedure to send a message or to receive a message. Below, in Figure 5, there is an example of a simple message passing application.

This program uses a producer-consumer pattern, where one thread produces a series of messages (a stream), and another thread consumes the messages. This pattern can be used to implement the observer pattern without deadlock risk and without the insidious error discussed in the previous section. The update procedure would always execute in a different thread from the observers, and would produce messages that are consumed by the observers. The code executed by the producing thread is given by the producer procedure and the code for the consuming thread by the consumer procedure. The producer invokes a procedure called send (to be defined) on line 4 to send an integer-valued message. The consumer uses get (also to be defined) on line 10 to receive the message. The consumer is assured that get does not return until it has actually received the message. In this case, consumer never returns, so this program will not terminate on its own.

```
1   void* producer(void* arg) {
2       int i;
3       for (i = 0; i < 10; i++) {
4           send(i);
5       }
6       return NULL;
7   }
8   void* consumer(void* arg) {
9       while(1) {
10          printf("received %d\n", get());
11      }
12      return NULL;
13  }
14  int main(void) {
15      pthread_t threadID1, threadID2;
16      void* exitStatus;
17      pthread_create(&threadID1, NULL, producer, NULL);
18      pthread_create(&threadID2, NULL, consumer, NULL);
19      pthread_join(threadID1, &exitStatus);
20      pthread_join(threadID2, &exitStatus);
21      return 0;
22  }
```

Fig. 5. Example of a simple message-passing application

4 Conclusions

To sum up, there has been an analysis of the design of the embedded systems and a focus on mid-level abstractions for concurrent programs. Examples of codes that handle multitasking have been demonstrated to provide a look on threads, which are sequential programs that execute concurrently and share variables and memory. However, threads are tricky to handle and require attention and study because there are several issues, such as mutual exclusion and deadlock. Message passing schemes avoid some of the difficulties, but not all of them, at the expense of being somewhat more constraining by prohibiting direct sharing of data. In the future, designers and programmers need to focus on using higher levels of abstraction (**see Figure 1**).

5 Acknowledgement

This research would not have been possible without the financial assistance of the following project: "Application of the mixed reality in the training and promotion of the cultural heritage for the purposes of the in the university information environment" financed by National Science Fund of the Ministry of Education and Science of the republic of Bulgaria with Contract № КР – 06 – ОПР 05/14 from 17.12.2018, led by Prof. DSc Irena Peteva.

6 References

[1] Berry G., Gonthier G. (1992). The Esterel synchronous programming language: Design, semantics, implementation. Science of Computer Programming, 19(2), 87–152. https://doi.org/10.1016/0167-6423(92)90005-v

[2] Burns A. Baruah S. (2008). Sustainability in real-time scheduling. Journal of Computing Science and Engineering, 2(1), 74–97.

[3] Chetto H., Silly M., Bouchentouf T. (1990). Dynamic scheduling of real-time tasks under precedence constraints. Real-Time Systems, 2(3), 181–194. https://doi.org/10.1007/bf00365326

[4] Graham R. L. (1969). Bounds on multiprocessing timing anomalies. SIAM Journal on Applied Mathematics, 17(2), 416–429. https://doi.org/10.1137/0117039

[5] Alur R., Dill D. L. (1994). A theory of timed automata. Theoretical Computer Science, 126(2), 183–235. https://doi.org/10.1016/0304-3975(94)90010-8

[6] Beyer D., Henzinger T. A., Jhala R., Majumdar R. (2007). The software model checker Blast. International Journal on Software Tools for Technology Transfer (STTT), 9(5-6), 505–525. https://doi.org/10.1007/s10009-007-0044-z

[7] Edwards, S. A., Lee E. A. (2003). The semantics and execution of a synchronous block-diagram language. Science of Computer Programming, 48(1), 21–42. https://doi.org/10.1016/s0167-6423(02)00096-5

[8] Wang Y., Lafortune S., Kelly T., Kudlur M., Mahlke S. (2009). The theory of deadlock avoidance via discrete control. In Principles of Programming Languages (POPL), ACM SIGPLAN Notices, Savannah, Georgia, USA, vol. 44, pp. 252–263. https://doi.org/10.1145/1480881.1480913

[9] Lamport L., (1977). Proving the correctness of multiprocess programs. IEEE Trans. Software Eng., 3(2), 125–143. https://doi.org/10.1109/tse.1977.229904

[10] Gamma E., Helm R., Johnson R., Vlissides J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley.

7 Authors

Radoslav Mavrevski is Chief Assistant in Department of Informatics, Faculty of Mathematics and Natural Sciences, member of University Center for Advanced, Bioinformatics Research, South-West University "Neofit Rilski", 66 Ivan Mihaylov Str., Blagoevgrad, Bulgaria. PhD on Informatics. Scientific Interest: programming, computer modelling, applied statistics and bioinformatics. He is one of the organizers of the South Eastern European Mathematical Olympiad for University Students (SEEMOUS) with International Participation, 2012, http://seemous2012.swu.bg/ and XXVII REPUBLICAN STUDENT PROGRAMMING OLYMPIAD, 2015, http://bcpc.eu/XXVII/.

Metodi Traykov is Assistant in Department of Informatics, Faculty of Mathematics and Natural Sciences, member of University Center for Advanced, Bioinformatics Research, South-West University "Neofit Rilski", 66 Ivan Mihaylov Str., Blagoevgrad, Bulgaria. PhD on Informatics. Scientific Interest: programming and bioinformatics. He is one of the organizers of the XXVII REPUBLICAN STUDENT PROGRAMMING OLYMPIAD, 2015, http://bcpc.eu/XXVII/.

Ivan Trenchev is Associate professor in Department of Electrical Engineering, Electronics and Automatics, Faculty of Engineering, member of University Center for Advanced, Bioinformatics Research, South-West University "Neofit Rilski", 66 Ivan Mihaylov Str., Blagoevgrad, Bulgaria and Associate professor in University of Library Studies and Information Technologies Sofia, Bulgaria. PhD on Informatics. Scien-tific Interest: virtual reality (VR), computer modelling and bioinformatics. He is one of the organizers of the XXVII REPUBLICAN STUDENT PROGRAMMING OLYMPIAD, 2015, http://bcpc.eu/XXVII/.

Article submitted 2019-02-08. Resubmitted 2019-03-25. Final acceptance 2019-05-09. Final version published as submitted by the authors.