Concurrent Program Verifier — a tool for teaching concurrent programming

Maxim Mozgovoy

Master's Thesis
Department of Computer Science
University of Joensuu, 2004

Abstract

This work firstly summarizes basic ideas, concepts and hardships of so-called concurrent programming. It also proves that concurrent programming is a very important branch of modern computer science, and it is worth studying. Then you can find a survey of existing educational software, intended for using in this area. Moreover, the paper provides a possible classification of such software packages. Special attention is paid to a new tool — CPV, which was designed and implemented mostly by the author of this work. This section also proves the necessity of CPV-like educational software. The next part describes various design decisions, which determine actual CPV possibilities and applications. After all you can find a description and evaluation of a small experiment on CPV usability, which results allows us to consider our software as a handy tool for teaching concurrency, suitable both for teachers and students.

Table of contents

1. Concurrent Programming Basics

To deal with concurrent (also known as *multithreaded*) programming, we should firstly consider this concept in more detail.

The instructions of any "ordinary" (non-concurrent) program are executed *sequentially*; therefore, non-concurrent programs are often referred as *sequential*. Unlike them, concurrent programs allow simultaneous (at least, illusive) execution of two or more instructions. We can define a concurrent program as a set of sequential programs which are executed in abstract parallelism [Ben-Ari90]. Each of these sequential programs will be now called *process*, while *program* is a whole set of them.

Note that the parallelism can be *abstract*: it is not necessary to have several physical processors for handling concurrency; instead we can share the power of the only one processor between different logical processes. To simulate concurrent behavior, single CPU can execute several instructions of the first process, switch to the second one, execute some its instructions and so on.

Here we can clearly see the difference between concurrent and *parallel* programming: the concept of parallel programming aims on achieving higher execution speed by utilizing several physical processors. Obviously, we will not reach any additional performance by running multithreaded application on a machine with single processor. Hence the aims of concurrent programming are different. Consider several examples which can clarify this question (ideas of examples are taken from [Feldman96]).

1.1. Time-sharing Operating Systems

Probably, any of us is now able to run several programs simultaneously on the one's PC. Right now I am typing this text using my favorite word processor; at the same time an audio player plays music, download manager downloads the latest Windows update from microsoft.com website, and a messenger allows me to stay online to able to contact my friends in case of need. I can also enable an antivirus monitor to protect my system from viruses and a firewall to ensure my Internet connection is safe. Each of these programs doesn't require a lot of processor time, so it is possible to utilize such "extra" time in a better manner than just executing an empty loop.

1.2. Real-time Systems

There are different kinds of computer software, existing to control some real physical systems. For instance, a computer can receive data from numerous electrical counters of the factory and maintain a log of their indications; computers can control railroad switches, automotive fuel systems and medical devices. These domains require the software to simultaneously gather some data from different places and process it; therefore, such software should be concurrent.

1.3. Modeling and Simulation

Many real-world simulations become easier to create in terms of concurrent programming, because our world is very concurrent itself. For instance, if you should simulate the behavior of an underground railroad, it is natural to consider each train as a separate process, interacting with other objects. When you are simulating the development of some biological population, and you need updated information to be on the screen each 5 seconds, it is easier to write a separate process, which performs all mathematical computations (and nothing

more), and another one — “a visualizer”, which takes current data and builds necessary graphs on the screen from time to time.

As we can see, the speed of execution can be not the main reason to divide the program into several different processes. Sometimes we can do it to get more accurate model of reality, sometimes we find concurrency to be an easy and a natural way for achieving desired functionality, and sometimes we are simply required to have a kind of multithreading inside the system.

Now we should consider a more practical question: well, we see the usefulness of concurrent programming, but how a concurrent program can be built with some popular programming language? The answer can be quite disappointing for somebody. For now, only a few languages provide possibilities to design concurrent programs in a standard, portable manner. And, probably, only two of them are widely known to the society: Ada [TDT97] and Java [AGH00].

The Ada programming language allows using special syntax for task-declaration sections:

```
task type MyTask is   -- declaration of task type
...
end MyTask;
task body MyTask is   -- definition of task body
...
end MyTask;

T1, T2 : MyTask;       -- create two processes of type MyTask
```

Thus, a concurrent program in Ada can be developed in a reasonably simple, natural way, by utilizing language syntax and semantics directly. Java doesn’t provide any specific language constructions for the developers of concurrent programs; instead, it offers a special type Thread, which we can extend to obtain an independent process:

```
class MyThread extends Thread {
    public void run() {
        // our process
        ...
    }
}

MyThread m1 = new MyThread();   // create two
MyThread m2 = new MyThread();   // threads
```

This simple example is demonstrative enough: a language, which gives us any possibilities for concurrent programming, usually provides some special data type for this purpose. The support of concurrent programming is a part of definition of both these languages — Ada and Java. An Ada compiler should be able to generate concurrent code on any platform; in case of

Java language the solution is even simpler: concurrency is supported directly by Java Virtual Machine. So, what about other languages? Probably, using any more or less modern language, it is possible to utilize an API, provided by underlying operating system. This solution is not portable, but it works, at least, on compatible operating systems. If OS API approach seems to be too low-leveled, you can try to find some additional library, providing more high-level objects and functions. For example, VCL, the standard library of Borland integrated development environments Delphi and C++ Builder, contains a special class TThread, more or less similar to Java Thread.

2. The Hardness of Concurrent Programming

As it is pointed in [Ben-Ari98], students often show misunderstandings of the very basic, cornerstone concepts of computer science, such as "variable", "parameter" or even "a computer" (or, from the constructivist point of view, the students have built consistent, but non-viable models of these concepts). Any kind of imperative programming introduces the ideas of variable, assignment and execution sequence; the paradigm of structural programming adds the concepts of procedure/function and user data type. OOP extends the world of concepts with classes, instances, methods, events and so on. Concurrent programming also adds its own, unique objects to this world; and, I believe, most of them are something, which is harder to understand than just a variable or a loop.

2.1. Understanding State Space Diagrams

To understand the basic difference between concurrent and sequential programming, let's consider a concept of *state space diagram.*

Our computers are finite machines, and a program is, basically, a sequence of instructions, transiting a computer from one state to another. If the execution of the program doesn't depend on some "outer" factors, such as random values generator, user input or external devices behavior, we can uniquely describe each situation in the program (in concurrent program, generally speaking) by a set of tuples:[1]

$\langle GV_1, \ldots, GV_n \rangle$ - global variables of the program

$\langle V_{1,1}, \ldots, V_{1,n1}$, LineID_1$\rangle$ - local variables of the 1st process and LineID

$\langle V_{2,1}, \ldots, V_{2,n2}$, LineID_2$\rangle$ - local variables of the 2nd process and LineID

...

$\langle V_{K,1}, \ldots, V_{K,nK}$, LineID_K$\rangle$ - local variables of the Kth process and LineID

LineID is an identifier of the current instruction[2] of the process (basically, an instruction number[3]). Since a sequential program is just a particular case of a concurrent one, we can use this notation for such programs also:

$\langle GV_1, \ldots, GV_n$, LineID$\rangle$

As long as we don't have different processes, our set of tuples degenerates into one element.

By using state space diagrams we can describe a behavior of any program (Ex. 1, Fig. 1; Ex. 2, Fig. 2).

Example 1. Simple sequential program

```
for i := 1 to 3 do
    f();
```

---

[1] If our program depends on outer factors, it can be also described in such manner, but these tuples will include additional parameters.

[2] which is supposed to be *atomic* (the processor cannot switch between processes while executing it).

[3] I will use also instruction itself as LineID, when such reference is unambiguous.

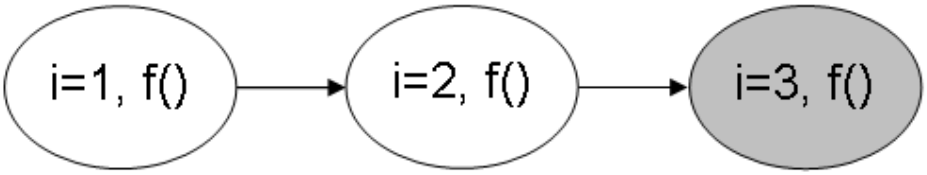


Fig. 1. State space diagram for Example 1 program

State space diagrams of sequential programs are simple: each state has one and only one outgoing transition. Therefore, the graph of state space diagram is, basically, a list. Now consider a state space diagram of some concurrent program.

Example 2. Simple concurrent program

```
process1: for i := 1 to 3 do
              f();

process2: for j := 1 to 3 do
              f();
```

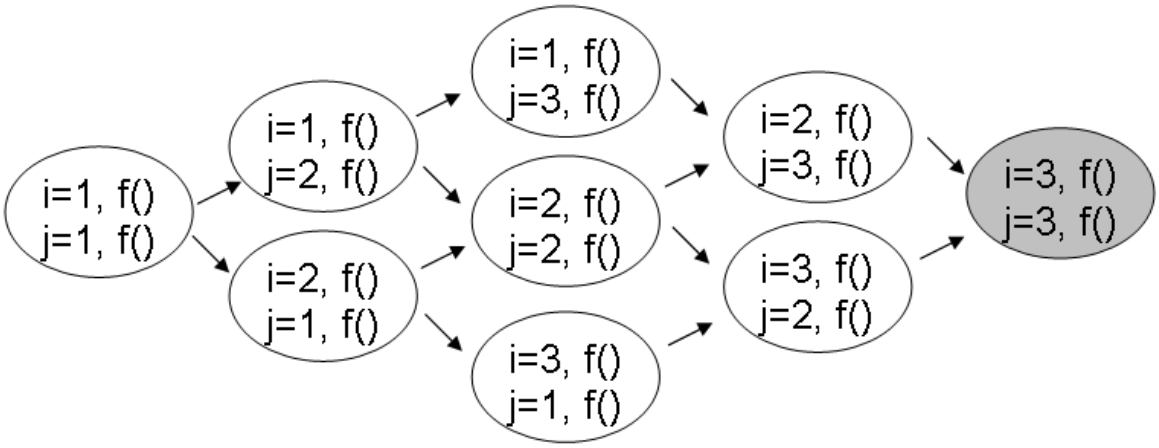


Fig. 2. State space diagram for Example 2 program

Note: in two last examples f() is assumed to be an atomic instruction, which also doesn't modify any variables.

As it can be seen, a state space diagram of a concurrent program has more complex structure. At any moment of time we cannot be assured which of the processes — process1 or process2 will change the current situation, so we should consider both possibilities.

From the observation of state space diagram complexity in case of concurrent program, we can make several conclusions:

1. There are several ways to execute one program; the behavior of the program may vary from one execution to another.
2. Different executions of the program can lead to different results.

3. Concurrent programs are hard to debug, since it is impossible to reconstruct the state of the program, which leads to an error (due to the same reason: we cannot control the way of execution of the program — it is defined by underlying OS).
4. Generally speaking, we should be able to obtain the same results regardless of actual execution flow. This observation is related to the concept of *correctness*, which is defined differently for sequential and multithreaded programs.
5. In general, OS scheduler may provoke incorrect behavior of our application. Therefore, we need a mechanism to constrain it.

State space diagram can be a powerful method of analyzing the behavior of the program. Later we will return to the role of state space diagrams in teaching concurrent programming. For now, just add state space diagrams to the collection of concurrency-related concepts.

At this moment let's consider in more detail an above stated question: how can we affect an actual execution of the program, to release it from the total control of the operating system?

2.2. Semaphores

Before proceeding, let's consider an example, where a freedom in OS actions can lead to an erroneous behavior.

Example 3. Prime-printing routine (error-prone)

```
GLOBAL N : Integer := 1;


process1: while true do
          begin
              Left := N;
              N := N + 1000 + 1;
              print_primes(Left, Left + 1000);
          end;


process2...processN: { the same }
```

Here you can see a skeleton of concurrent prime-finding routine (which prints prime numbers in the range [1…+∞)). Suppose we have a multi-processor machine (or a cluster) and we want to utilize existing hardware. The suggested approach is simple. Maintain a global variable N, which indicates the left border of unexplored range of values. Each process (running on the separate processor) memorizes the current value of N and performs an assignment

```
N := N + 1000 + 1;
```

which means "I am responsible for the analysis of the next 1000 numbers". After that this process analyzes its range, prints found primes and requests the next interval to be explored.

At the first sight this algorithm seems to be simple and correct: obtain next interval, shift the border value, process the interval and repeat. But a closer look can reveal a scenario, which leads to an error:[4]

```
process1: Left := N;                          // N = 1, Left := 1;
process2: Left := N;                          // N = 1, Left := 1;
process1: N := N + 1000 + 1;                  // N := 1002
process2: N := N + 1000 + 1;                  // N := 2003
process1: print_primes(Left, Left + 1000);    // print_primes(1, 1001);
process2: print_primes(Left, Left + 1000);    // print_primes(1, 1001);
```

In other words, both processes firstly read the same value of N, and then increase it. As a result, each of these processes will analyze the same interval [1…1001], and, moreover, the next interval [1002…2002] will remain unconsidered at all!

The standard way of solving such problems is to use *semaphores* [Dijkstra68]. As it is defined in [Ben-Ari90], a semaphore (S) is a non-negative integer variable with only two defined atomic operations:[5]

Wait(S): if S > 0 then S := S – 1 else block the process.

Signal(S): if there are any blocked processes on this semaphore, release one of them else S := S + 1

Consider now a code fragment:

```
GLOBAL S : Semaphore := 1;  // initial value


process1: //...
          Wait(S);
          f1();
          f2();
          Signal(S);


process2: //...
          Wait(S);
          g1();
          g2();
          Signal(S);
```

[4] Note that here I am assuming a so-called *Load-Store model*, which supposes instructions, reading and writing global variables as the only atomic.

[5] Strictly speaking, any semaphores are beyond the Load-Store model. From the theoretical point of view, there is a possibility to solve our problems remaining inside this model (by using so-called Dekker's algorithm), but from the practical side there is no reason to do it.

When one of the processes executes the statement Wait(S) (there will be no process switching during the execution of this instruction due to its atomicity), another one cannot enter this Wait(S)…Signal(S) section. After first Wait(), the value of the semaphore will be set to zero, so any other process will be blocked during an attempt to execute this operation. An execution of Signal(S) operation releases one of suspended processes (in our example there are only two processes at all, so it just releases *another* process), and then it can continue working. Hence the number of possible scenarios reduces to two:

Scenario 1.

```
process1: Wait(S);
process1: f1();
process1: f2();
process1: Signal(S);
process2: Wait(S);
process2: g1();
process2: g2();
process2: Signal(S);
```

Scenario 2.

```
process2: Wait(S);
process2: g1();
process2: g2();
process2: Signal(S);
process1: Wait(S);
process1: f1();
process1: f2();
process1: Signal(S);
```

So, semaphores can be used to group a set of instructions into one *atomic block*. Such blocks (between Wait(S) and Signal(S) are often referred as *critical sections*).

The situation can be also illustrated by a real-world example with *real* semaphores and locomotives (Fig. 3).

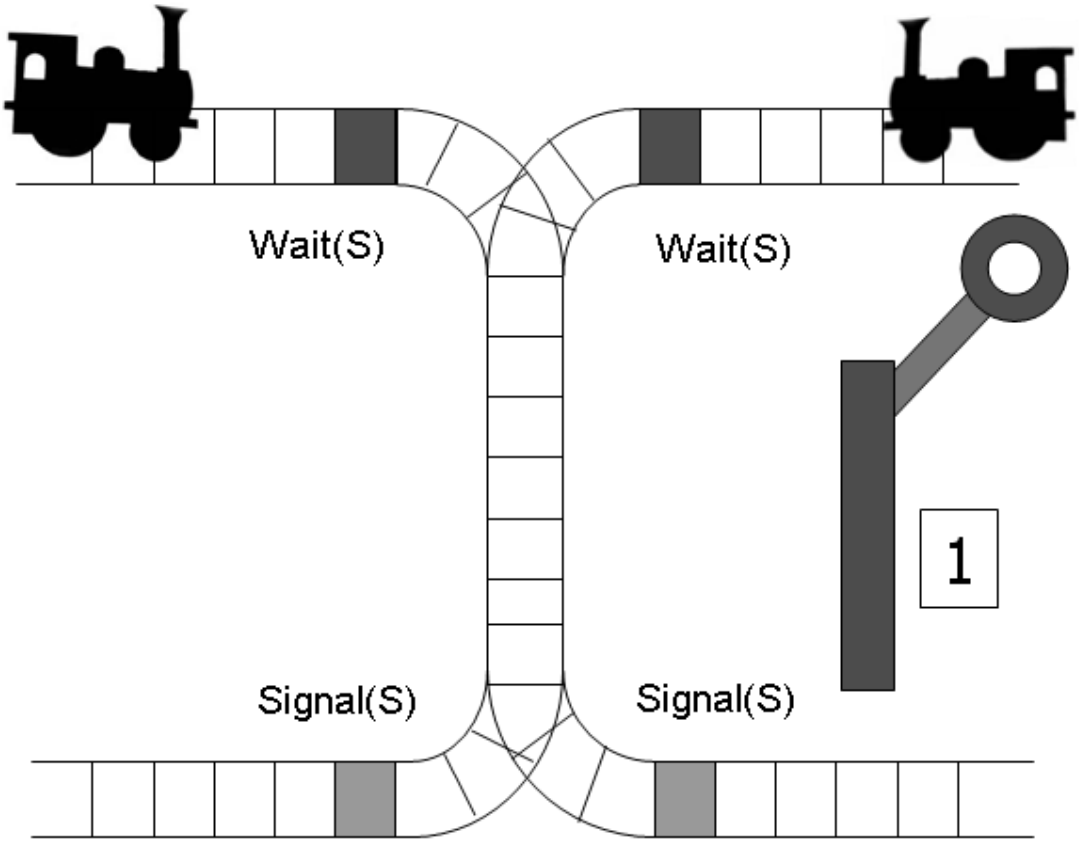


Fig. 3. Real-world example of semaphore usage (1)

When any locomotive passes through Wait(S) point, the semaphore blocks any other attempts to do it (Fig. 4). Now a semaphore allows passing only after Signal(S) command.

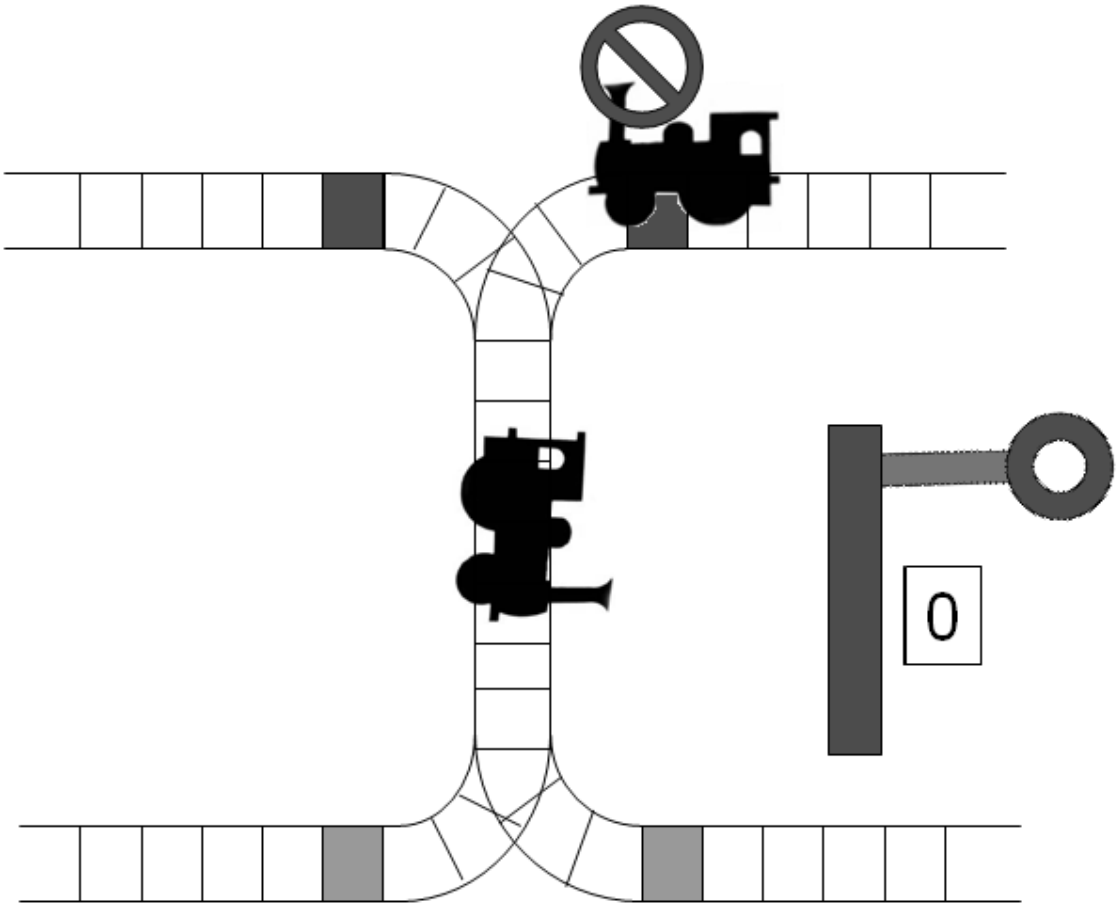


Fig. 4. Real-world example of semaphore usage (2)

Now we can return to the example of prime printing routine and rewrite it using the semaphore:

Example 4. Prime-printing routine

```
GLOBAL N : Integer := 1;
       S : Semaphore := 1;


process1: while true do
          begin
              Wait(S);
              Left := N;
              N := N + 1000 + 1;
              Signal(S);
              print_primes(Left, Left + 1000);
          end;
```

```
process2...processN: { the same }
```

Note that only Wait(S)…Signal(S) section is protected. An OS still can perform task switching during the execution of print_primes() procedure.

Semaphores can be used not only for critical sections organization; they are intended to be a tool, suitable for applying in various problems, where a synchronization of different processes' actions is needed.

As usual, any kind of additional possibilities cause additional responsibility. Once decided to affect the execution of the program, we should be ready for the new obstacles. Semaphores allow processes suspend an execution of other processes, and this possibility is intrinsically error-prone.

One of the common troubles is so-called *deadlock*, a situation, when some process A blocks process B, and at the meanwhile, the process B locks A. The program will obviously reside in this state forever, since no way out can be found. The typical example of the deadlock is the following sequence of database operations:

Example 5. Deadlock scenario

```
GLOBAL S1: Semaphore := 1;
GLOBAL S2: Semaphore := 1;
...
                    // process1 wants to write Table1
process1: Wait(S1); // lock Table1
                    // process2 wants to write Table2
process2: Wait(S2); // lock Table2
                    // process1 wants to write Table2
process1: Wait(S2); // trying to lock Table2; suspended on the semaphore S2
                    // process2 wants to write Table1
process2: Wait(S1); // trying to lock Table1; suspended on the semaphore S1
// process1 waits until process2 unlocks Table2
// process2 waits until process1 unlocks Table1
```

The conclusion is simple: we should pay additional attention (up to formal verification) when working with semaphores.

3. Teaching concurrent programming

Previous passages described the basics of concurrent programming (very shortly, since this work deals with concurrent programming, but isn't dedicated to it). Now we can see that such kind of programming is a very specific branch of computer science with its own concepts, idioms, methods and even standard problems (such as "Readers and writers" or "Dining philosophers", [Whiddet87]). Undoubtedly, these specific points require quite specific approach to the teaching process and specific educational tools. We will concentrate on the above described concepts (although, they are just a top of an iceberg):

1. ways of program execution, state space diagrams
2. the concept of correctness
3. semaphores and semaphore-related problems

There are many textbooks on concurrent programming available — both theoretical and practical-oriented ([MK99], [Snow92], [Hansen02], [Lea99]). The detailed discussion of approaches for the teaching is beyond of the scope of this work; instead we will consider *software tools*, which can be used by a teacher.

3.1. Software Robots

The first program of this sort I am familiar with is CRobots, written in a quite distant nowadays year 1985. The basic idea of this program is to provide a nice competition for the programmers: you are supposed to program a behavior of a "software robot" by using some special API. Basically a robot can examine an environment, turn, move, and fire its weapon. Then these programmed robots are gathered in a "playground" and the battle begins. The survivor is a winner of the competition.

During the ensuing years such CRobots-like programs were noticeably developed. Among "bells and whistles" like nice SVGA graphics or network play support we obtained also the possibility of using object-oriented approach to program robots, special environments for designing robots and a support of some other programming languages, such as TCL or Java (Fig. 5, Fig. 6).

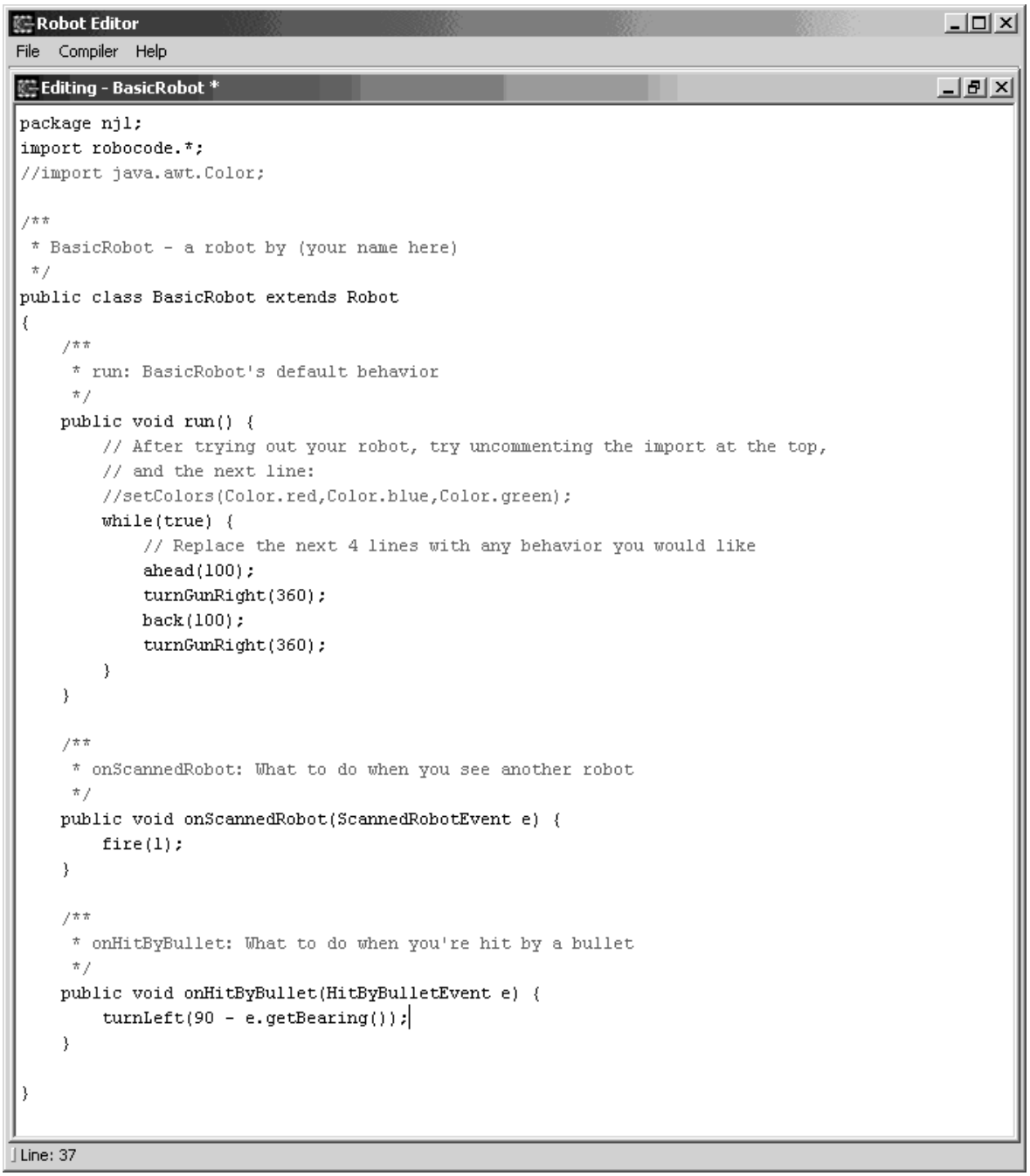


Fig. 5. IBM's Robocode IDE

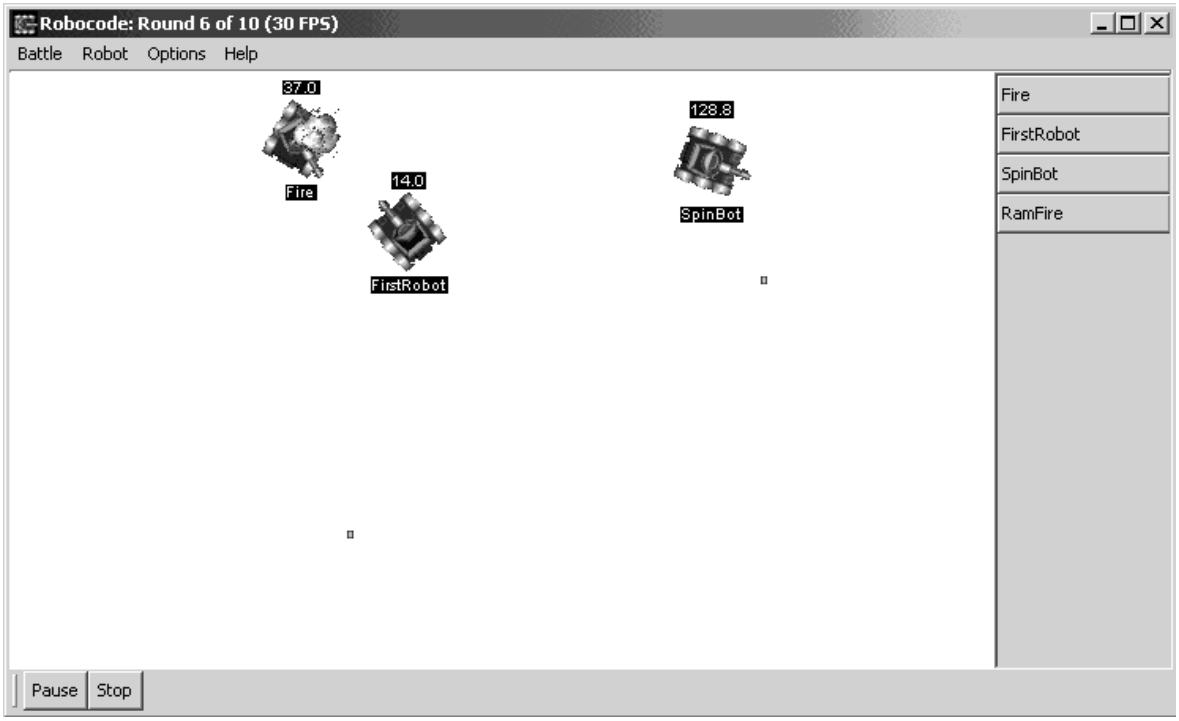


Fig. 6. IBM's Robocode playground

Many modern robot-designing environments are meant not for experienced programmers' fun, but for providing an exciting starting point for the beginners. You can write just 10-15 lines of more or less simple code, and you obtain a personal robot, which works exactly as you specified. Surely, it is more interesting than finding a maximum array element or implementing bubblesort!

I should mention that not all roboworld simulators are aiming at deathmatch play (which is closer to an AI development). Some of them really intended to teach programming by providing a kind of simple and visual programming language; maybe it is legal to consider them as highly upgraded LOGO ([Yoder90]) versions. A book [BSRP97], which describes the world of Karel J. Robot simulator, explicitly claims the possibility to use this simulator as an environment for studying concurrency. The book contains several examples of "concurrent robots"; it shows how to illustrate some basic concepts of concurrent programming, such as simultaneous execution, so-called "race conditions", deadlocks, etc. by means of roboworld. Ex. 6 contains a listing of the simplest concurrent robot program, which works in a Karel J. Robot world.

Example 6. Concurrent robots

```
class Racer extends Robot {
    public Racer(int Street, int Avenue, Direction direction, int beepers) {

        super(Street, Avenue, direction, beepers);
        World.setupThread(this);
    }
    public void race() {
```

```
        while(!nextToABeeper())
            move();
        pickBeeper();
        turnOff();
    }
    public void run() {
        race();
    }
}
...
public static void main(String [] args) {
    Racer first = new Racer(1, 1, East, 0);
    Racer second = new Racer(2, 1, East, 0);
}
```

Robots in this example can race each other to some goal. The first racer begins on the 1st street, 1st avenue; the second one on the 2nd street, 1st avenue. Both robots face east; somewhere in front of each robot is a beeper. Next, we start them simultaneously to see who will be the winner.

Links to Software Robots Related Resources:

IBM's Robocode: http://robocode.alphaworks.ibm.com/home/home.html

CRobots: http://www.nyx.net/~tpoindex/crob.html

TCLRobots: http://www.nyx.net/~tpoindex/tcl.html#TclRobots

Karel J. Robot: http://csis.pace.edu/~bergin/karel.html

3.2. Concurrency Simulators

The tools, more directly related to multithreaded programming, are *concurrency simulators*. As noted in [Persky99], it can be difficult to get familiar with concurrent programming by using real programming languages from the very start. At first, concurrent systems development often requires knowledge of internal OS API (at the beginning of this work I've mentioned that only a few languages provide possibilities to design concurrent programs in standard, portable manner); at second, we cannot see details of an actual execution scenario, cannot perform step-by-step trace of a real multithreaded program.

The popular solution suggests using a "concurrency simulator" — special software, which allows us to create, to trace and to retrace arbitrary execution scenarios of concurrent programs.

One of the most (if not the most) popular concurrency simulator at the present time is BACI. Its webpage says that BACI is used in dozens of schools, colleges and universities all over the world. Current BACI package includes two compilers of simplified concurrent languages — C-- and Concurrent Pascal, and an actual simulator. The simulator, basically, executes compiled program (Ex. 7).

Example 7. BACI at work

```
// listing of concurrent program

const int m = 5;
int n;


void incr (char id)
{
    int i;


    for (i = 1; i <= m; i = i + 1)
    {
        n = n +1;
        cout << id << " n =" << n << "  i =";
        cout << i << " " << id << endi;
    }
}


main()
{
    n = 0;
    cobegin
    {
        incr( 'A' ); incr( 'B' ); incr( 'C');
    }
    cout << "The sum is " << n << endl;
}

//-----------------------------------------

...

// resulting printout

Source file: incremen.cm Wed Oct 22 21:18:02 1997
Executing PCODE ...
C n =1 i =A n =1 C2 i =
1 A
C n =4 i =2 C
B n =A n =5 i = 24 A
  i =1 B
AC n = n =6 i =3 C6 i =3
A
C n =7 i =4 C
B n =9 i=2 BA n =8
  i =4 A
C n =8 i =5 A n =9C
  i =5 A
B n =10 i =3 B
B n =11 i =4 B
```

```
B n =12 i =4 B
The sum is 12
```

After BACI a BACI Debugger was released. It allows user to trace concurrent programs. Next steps of the development of this simulator led to the creation of GUI for BACI (unfortunately, works only under *nix systems) and jBACI (Fig. 7).

jBACI is a new concurrency simulator, written in Java. Actually, it is more than just a simulator — jBACI is a complete IDE, which includes editor, necessary compilers (taken from BACI package) and visual debugger. It is possible to use graphical primitives in jBACI — this nice feature enables us to write more demonstrative programs.

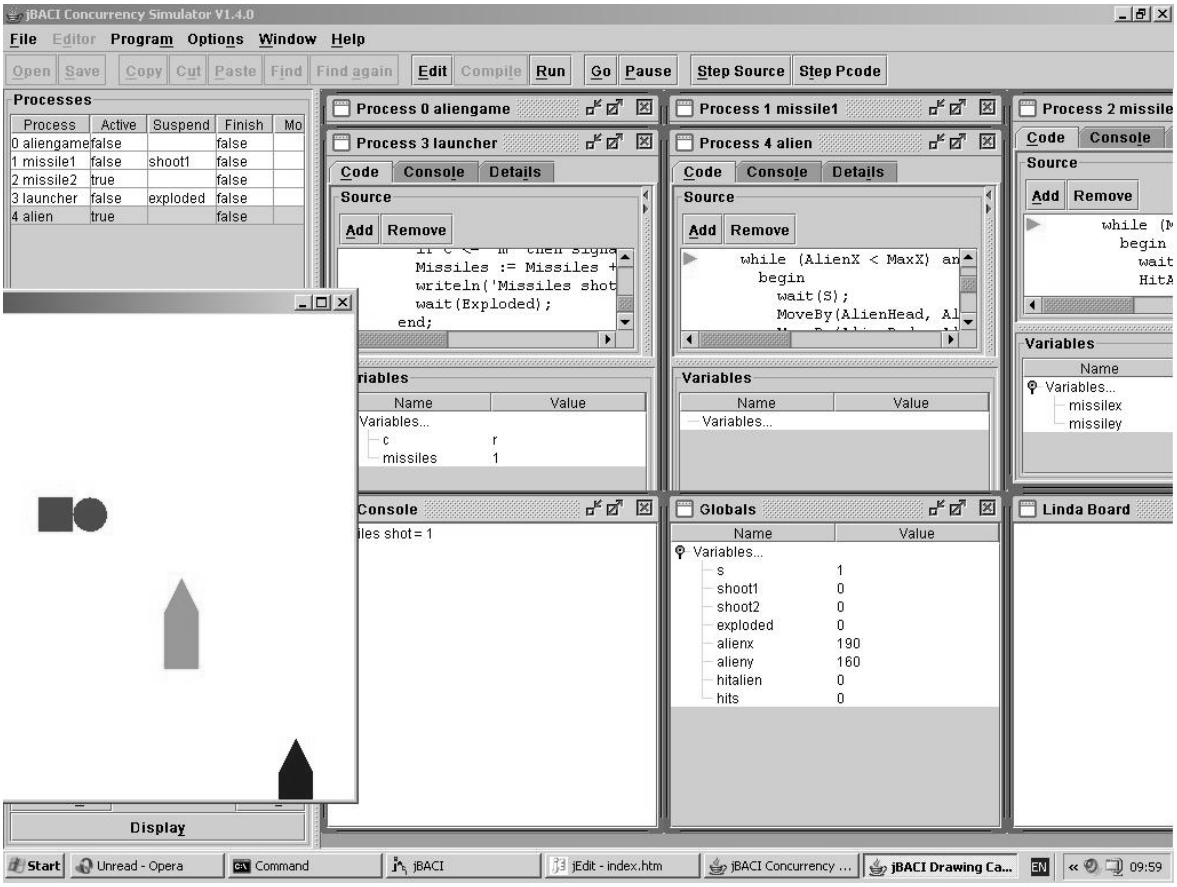


Fig. 7. jBACI concurrency simulator

Except BACI variations, I can mention SimAda concurrency simulator, written in 1999 by Yakov Persky. It was developed using Ada 95 programming language and can execute programs written in "SimAda Language" — a special Ada subset.

Links to Concurrency Simulators:

BACI: http://www.mines.edu/fs_home/tcamp/baci/

jBACI: http://stwww.weizmann.ac.il/g-cs/benari/jbaci/

SimAda: http://stwww.weizmann.ac.il/g-cs/benari/files/simada.zip

http://stwww.weizmann.ac.il/g-cs/benari/files/ypthesis.zip

3.3. Model Checking Software

Probably, the only one way to be more or less assured in validity of your software, known to every programmer, is thorough testing. People develop various techniques for testing (see, for example, [Beck99]), write scientific, near-scientific and technical papers, explaining, how to perform testing in the right way, how to select input data and analyze results. Although, as Dijkstra noticed, program testing can be used to show the presence of bugs, but never to show their absence, testing still remains the most popular and widely used nowadays. I think, really well-performed testing is a quite sophisticated procedure, and alternative approaches are even trickier. But sometimes we have no choice (see also [Ben-Ari00], [KP99]):

1. The device, intended to run our program has no capabilities for debugging and testing software (a rocket, a microchip).
2. The software is too complex and the ways and conditions of its usage are unpredictable (operating systems).
3. The conditions of testing are unrepeatable due to specific hardware. For example, if our application utilizes two processors of a multiprocessor machine, we cannot control which processor will be used for executing one or another algorithm of an application. It is extremely hard to catch a bug in this situation if it is caused by a processor malfunction ([KP99]).
4. The conditions of testing are unrepeatable due to a huge amount of possible ways of program execution.

The last reason is the most interesting for us now, since it directly concerns to concurrent programming. If you gave numbers (1, 2, 3) as an input arguments to a sequential program and obtained 6 as a result, this is probably means that next time you will also obtain 6 for the same arguments (surely, if your program doesn't depend on some random values or user input). For concurrent program such "testing" means nothing: next time an operating system can select another execution scenario and the result can be completely different. The mirror example of the same situation is even more offensive: suppose you found an erroneous behavior of your program. On some single run it produces 5 for the arguments (1, 2, 3). But all next runs produce 6 — the correct result — and you are unable to find the source of an error, since only one or several scenarios among hundreds and thousands can cause it.

An alternative approach to usual testing (which is basically a kind of run-and-see-what-will-happen technique) called *formal verification*. This term stands for proving or disproving the *correctness* of a system with respect to a *specification*, using mathematical methods. In practice it often means formulating some *properties* of your application by means of *linear temporal logic* or *computational tree logic* and using specific verification software which analyzes your program and tries to make conclusions.

A small efficiency of an ordinary testing in concurrent programming increases the importance of formal verification methods. No wonder that many such "auto-verifiers" are aiming more at concurrent and distributed, not at sequential systems.

All these observations entail the following conclusion: since formal verification methods are so important in the world of concurrent programming, a good course on this subject should include, at least, their basics.

There are several enterprise-level verification software packages available.

One of the most popular and well-known among them is Spin (and its recent convenient frontend jSpin) — a tool, which was developed in 80s at Bell Labs, and has been freely

available since 1991 [Holzmann03]. Spin uses a special high level language PROMELA to specify system properties. Once described your application, you can invoke Spin to perform checking for the logical consistency of a specification. The tool reports about found deadlocks, race conditions and other common obstacles of concurrent programs.

The simplest example of PROMELA usage is ordinary assertions:

```
assert (NumberOfLeaders == 1)
```

This sentence tells to Spin that the value of NumberOfLeaders variable should be equal to one at this moment. In case of a violation of this condition, Spin will report an error.

More powerful feature of PROMELA is the support of temporal logic assertions. Temporal logic works with propositions that change with time. For instance, if prefix <> means "eventually" and [] — "always", we can express the assertion "NumberOfLeaders should eventually become equal to 1 and remain in this state forever" in a following manner:

```
<>[](NumberOfLeaders == 1)
```

Similar aims and comparable possibilities can be found in other systems — SMV and TLV. The main difference between them is a way of describing properties: SMV uses computational tree logic while TLV utilizes linear temporal logic (like Spin).

Among these serious systems I should mention STeP (stands for Stanford Temporal Prover) [BBCCKMSU96]. It works with hardware or software descriptions, expressed as transition systems and temporal logic formulas. STeP uses verifications rules and diagrams, auto-generated invariants, model checking and decision procedures for software verification.

A different enough approach is used in Uppaal — a very interesting system, designed jointly in Aalborg (Denmark) an Uppsala (Sweden) universities (Fig. 8, [BL96]).

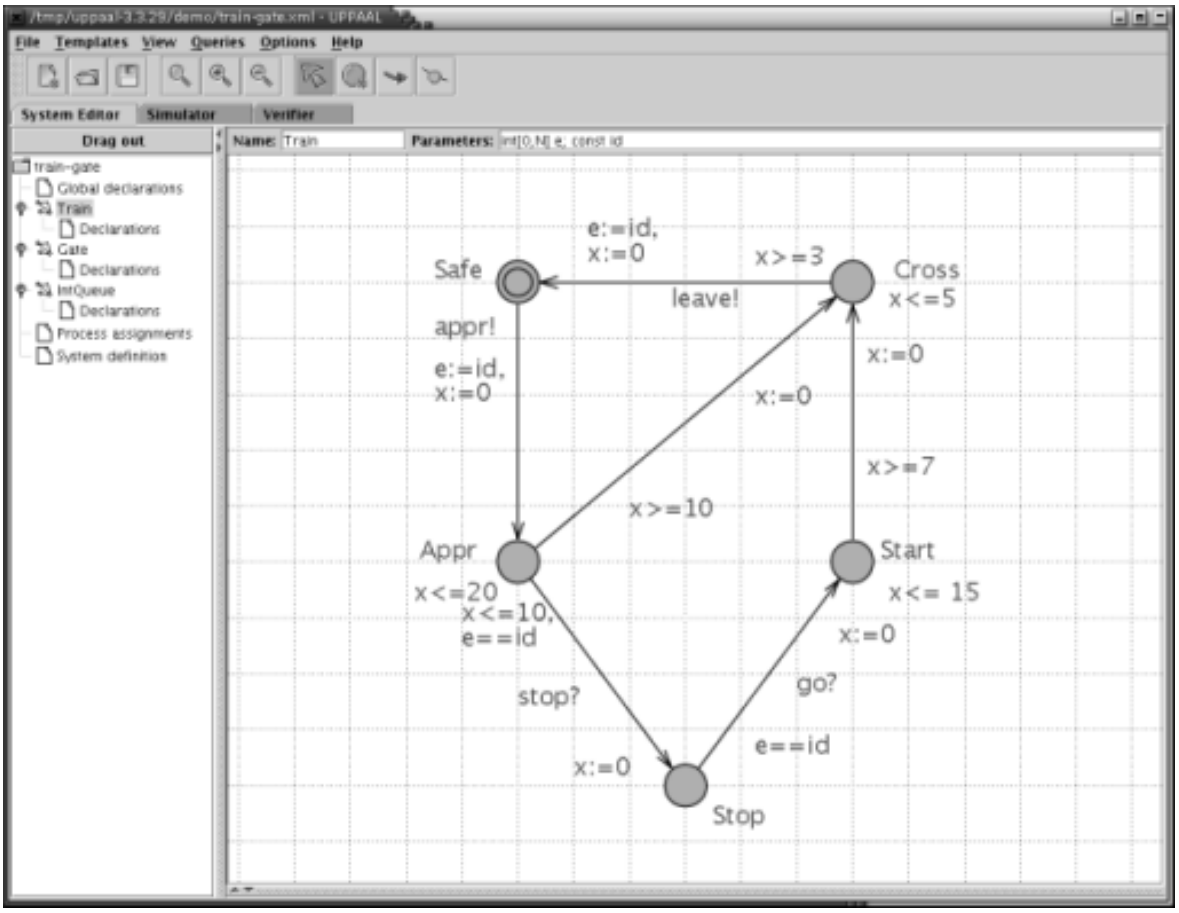


Fig. 8. Uppaal IDE

Uppaal uses a special guarded command language with data types to describe the program (as a network of automata) to be analyzed. Then a simulator can examine possible executions of your system. Uppaal also includes a model checker, which can check invariants and perform reachability analysis of the program.

The "visuality" of the system (its authors explicitly underline the ease of usage as one of the main design criteria) makes it suitable for education purposes.

Links to Model Checking Software:

Spin: http://spinroot.com/spin/whatispin.html

jSpin: http://stwww.weizmann.ac.il/g-cs/benari/jspin/

jBACI: http://stwww.weizmann.ac.il/g-cs/benari/jbaci/

SMV: http://www-2.cs.cmu.edu/~modelcheck/smv.html

TLV: http://www.wisdom.weizmann.ac.il/~verify/tlv/index.shtml

STeP: http://www-step.stanford.edu/

Uppaal: http://www.docs.uu.se/docs/rtmv/uppaal/

4. CPV (Concurrent Program Verifier)

With high respect to all above described systems, I believe, there is a place under the sun for one more modest tool for teaching concurrent programming — CPV, which 1.0 version is written by me under supervision of prof. M. Ben-Ari.

Probably, before deciding to develop any new educational tool, a researcher should first ask oneself: what kind of software a teacher needs? What kind of software a student needs? Why existing software solutions are not sufficient?

Even if the different researchers' answers are very similar, it is better to have two educational systems than one: the possibility to select is generally a very good thing.

In our case (we are aiming at basic-level teaching) I can underline the following points:

1. A teacher needs a tool, which can be used as an addition to ordinary slides for teaching *basics* of concurrent programming.
2. This tool should be as *visual* as possible, so it can be utilized during lectures.
3. The software should be *easy to use*; it also should not require specific *knowledge* or unusual *skills* from the teacher.
4. Student's needs are like a reverse of the medal. We want to develop a system, which could be used as an addition to the lecture notes (so student can use the tool to gain a better understanding of material) and as a nice companion during homeworks. Surely, we try to require as less experience and specific knowledge as possible from students.

I have mentioned about "teaching basics" of concurrent programming. No doubt, different courses make emphasis on different points of concurrency, so these "basics" may vary. Educational software tools very seldom show themselves as "pure": almost always we can see the direction of the authors' courses. For instance, I have doubts about studying the world of Karel J. Robot just to use it for demonstrating concurrency. The authors of Karel J. Robot use their tool for various purposes, and concurrency teaching is only one of them. Similarly, I believe, the usage of automata in Uppaal is not accidental: it is very likely that automata theory plays an important role in the courses of Aalborg and Uppsala universities.

Our approach also falls under influence of our interests. For example, in certain applications it is very important to be able to control different threads: to create them, to communicate via messages, to suspend or destroy if necessary. As I told before in "Teaching concurrent programming" section, we are more interested in rather theoretical, low-leveled concepts:

1. The process of concurrent program execution. Different execution paths.
2. State space diagrams as a method of *visualization* and *verification*.
3. Basic concurrency-related concepts: *semaphores, mutual exclusion, deadlocks*, etc.

The core idea is simple: since state space diagrams is a universal method of visualization and verification, we can try to develop a tool, which builds a state space diagram for any given program. Although, details make a program in reality; details can spoil any excellent idea and details can turn even a weak basis into something impressive. That's why I'd like to explicitly substantiate the rationale for our design decisions. I will also provide some technical aspects (e.g. the architecture of the system) to make this description more complete.

As we are talking about analyzing an execution of the *program*, it is rationally firstly to define how we can describe a program.

One way is quite obvious: use some existing programming language or create your own. Generally speaking, such solution provides the highest expressive power for the user. No wonder, serious simulators like Spin or TLV use this approach. On the other hand, "textual" languages are not visual; moreover, to introduce any such language means to enforce a teacher/student to learn "yet another programming language". In case of Spin this is not important, since Spin is intended to be a serious long-term instrument for serious people. If you really decided to use Spin at the work, there should be no problems to dedicate several days to PROMELA studying; but our situation is quite different: the need of dealing with some new programming language can discourage many people.

An example of an alternative approach is shown in Uppaal: a text-based programming language is not the only way to program! Uppaal's "automata language" is very specific, that's why we cannot use it in our software, but in any case I can point, at least, three disadvantages of any "graphic language" of such kind:

1. Usually, an expressive power of graphic languages is lower in comparison with ordinary ones.
2. Graphic programs require more space.
3. The process of drawing is considerably slower than typing; therefore, for an experienced programmer it is easier to ***write*** a program than to ***draw*** it (although, for a newcomer, who is not yet familiar with a syntax, it is not so).

Anyway, for educational software, aiming at small program analysis, such graphic language can be a good choice. Surely, I am talking not about graphic automata representation, "robot control language" or something like that; it should be a simple, very well-known concept.

We believe that so-called ***flowcharts*** are good candidates for representing our sample programs. Flowchart is a very common "graphic language", which uses different blocks, connected by arrows (Fig. 9). At least, as I remember, I studied flowcharts at school before any ordinary programming language.

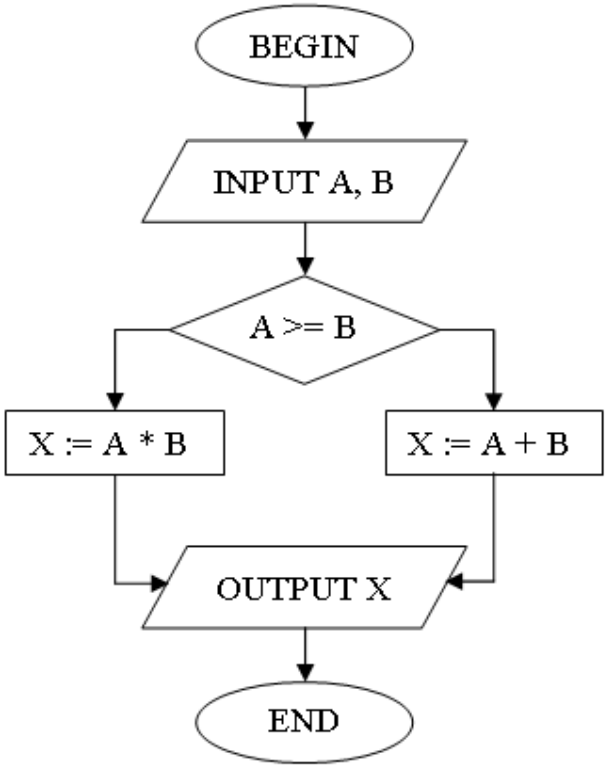


Fig. 9. Flowchart example

After language selection, we can discuss visualization module. A naïve approach assumes the usage of straightforward architecture:

`Flowchart Editor à Visualization Module`

But after a closer look this scheme shows serious disadvantages. At first, it is enough complicated to create a simulator, which directly executes flowcharts. A standard solution in such situations is to use a kind of "intermediate language". For example, a C++ compiler can work according to the following scheme:

`C++ à C à Assembler à Object code`

It is much easier to convert C++ program to a C analogue than to compile it directly, and C program can be converted to Assembler statements instead of straight compilation also. Only at Assembler level you have no choice: next step is machine code. This approach also allows us to utilize existing Assembler compiler when developing a C translator and existing C compiler during design of C++ one.

The second disadvantage is a high level of integration of two logically different modules of the program — Flowchart Editor and Visualization Module. Generally speaking, any modules should be bound to each other as weak as possible — it makes program logic clearer and simplifies maintenance.

That's why I've decided to use slightly more complex architecture (and now I think this decision noticeably simplified the process of development):

`Flowchart Editor à Intermediate Language à Visualization Module`

Let's have a closer look on the design principles of these elements.

4.1. Flowcharts in Concurrent Programming

There are not so many modifications we should introduce to ordinary school flowcharts to adapt them for our needs.

In traditional flowchart a *variable declaration section* is almost always omitted. You just use expressions like X := 5 without explicit X definition. On the other hand, probably any model checking software requires you to specify a domain for every variable. If you clearly understand the logic of your program, this action will take only a bit of time, while advantages are more than just noticeable, since a simulator can catch and report out-of-range errors. In case of state space diagram building the usage of such constrained variables is crucial. Consider, for example, the following (probably, senseless) fragment:

```
while true do
    i := i + 1;
```

Theoretically, the value of i should increase to infinity, but in practice (since our computers are finite machines) we should stop at some point. The only question is when? After 1000 generated states? 65535? 2^{32}? To specify a variable domain mean to tell simulator: "I know what I am doing" and to decide this question. A simulator should perform state generation while every variable is inside its range.[6]

These observations lead us to the conclusion: each variable in CPV should be typified and constrained; a variable should also have an initial value to exclude any random factors while building a state space diagram.

Another "variable issue" concerns the fact that in concurrent programming we have global and local (in a certain process) variables. It should be also taken into account.

Suppose we have a special "declaration area" in any process and a "global declaration area" in the program. Now we can define variable declaration syntax:

Type	Syntax	Example
Integer	Name : integer(MinValue..MaxValue) := InitValue;	a : integer(-5..100) := 5;
Boolean	Name : boolean := InitValue;	flag : boolean := true;
Semaphore	Name : semaphore := InitValue;	S : semaphore := 3;

As it can be seen, in CPV you should supply an exact range for every integer variable. In most programming environments variable type implicitly specifies its range: for instance, char almost always means [0..255], while unsigned int stands for [0..2^{32}-1]. Such syntax in CPV is forbidden: we require an explicit specification of the inverval, so that we can ensure that only a very few values are possible for each variable, and thus the state diagram will be small enough to be displayed.

I should mention that "semaphore" in our case is a general semaphore.

Next question is related to flowchart blocks. After several discussions we stopped on the following types:

Type	Description

[6] Nevertheless, it makes sense to stop analysis also if the state space diagram becomes really huge.

Assignment	Ordinary assignments like A := B, X := Y + 1, etc.
Semaphore op.	Wait() and Signal() operations.
Branching	Conditionals: A < B, Z >= 5 and so on.
End	End-of-process indicator.

There is also a possibility to *mark* any existing block as starting.

It is not enough to define block types. Then we should agree what to put inside these blocks. In our case it is a set of Pascal-styled statements:

Statements for Assignment Blocks

Syntax	Examples	Comment

Lvalue := Rvalue;	a := b;
	flag := true;
	x := 15;
		Simplest assignment: works both for integer and Boolean variables. Lvalue is a variable, Rvalue is a variable or constant of the same type (integer or Boolean).

Lvalue := not Rvalue;	flag := not flag; end := not false;	Assignment with NOT operation. Works with Boolean values only. Lvalue is a Boolean variable, Rvalue is a Boolean variable or constant.
Lvalue := Rvalue1 + Rvalue2;	addr := base + offset; i := i + 1;	Assignment with "+" operation. Works with integer values only. Lvalue is an integer variable, Rvalue1 and Rvalue2 are integer variables or constants.
Lvalue := Rvalue1 - Rvalue2;	num := num – 1; weight := full – cargo;	Assignment with "-" operation. Works with integer values only. Lvalue is an integer variable, Rvalue1 and Rvalue2 are integer variables or constants.
Lvalue := Rvalue1 and Rvalue2;	result := op1 and op2;	Assignment with AND operation. Works with Boolean values only. Lvalue is an integer variable, Rvalue1 and Rvalue2 are Boolean variables or constants.
Lvalue := Rvalue1 or Rvalue2;	Flag := flag or mask;	Assignment with OR operation. Works with Boolean values only. Lvalue is an integer variable, Rvalue1 and Rvalue2 are Boolean variables or constants.

Statements for Semaphore Blocks

Syntax	Examples	Comment
wait(SemaphoreName);	wait(S);	Standard semaphore operation Wait().
signal(SemaphoreName);	signal(S);	Standard semaphore operation Signal().

Statements for Branching Blocks

Syntax	Examples	Comment
value1 op value2, where op is =, <>, <, <=, > or >=	a = 5 height <> width flag <= true	Condition test: value1 and value2 are variables or values of the same type (integer or Boolean). Boolean values can be used only for equality and inequality test.

Our flowchart editor realizes these principles (Fig. 10). Note that different processes are shown on different flowcharts.

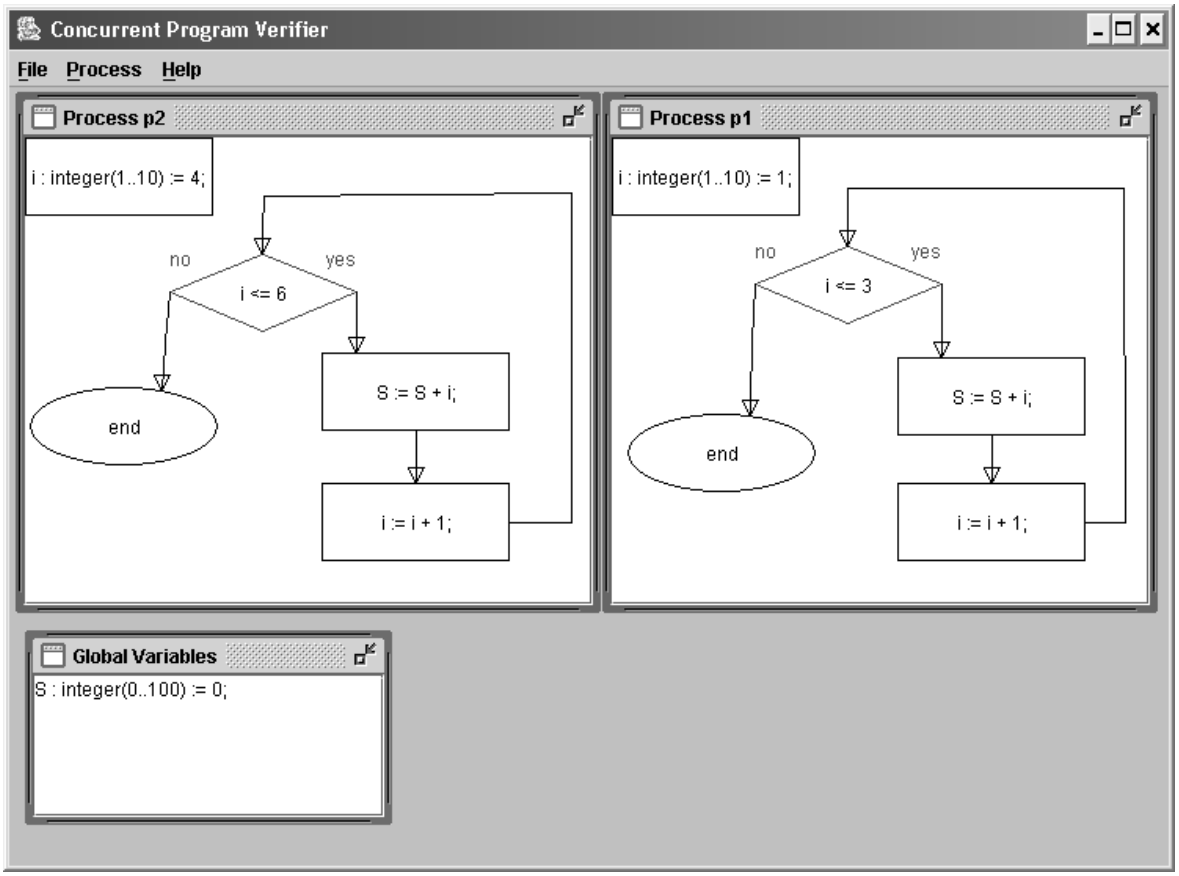


Fig. 10. Flowchart Editor

4.2. Intermediate Language

After you command to begin state space diagram building, CPV firstly performs a flowcharts-to-intermediate language translation. Intermediate language has the same expressive power as flowcharts, but it is much easier to execute. Consider various IL elements:

IL Element	Corresponding Flowchart Element	Description
__commonvariables	—	Beginning of the common variables declaration section.
__endofcommonvariables	—	End of the common variables declaration section.
__process ProcName	—	Beginning of the process ProcName and its local variables declaration section.
__code	—	End of the local variables declaration section, beginning of the actual code.
__endproc	—	End of the process.
bool x Value	x : boolean := Value;	Boolean variable declaration.
int x Min Max Value	x : integer(Min..Max) := Value;	Integer variable declaration.
sem x Value	x : semaphore := Value;	Semaphore variable declaration.
asgn x y goto N	x := y;	Simple assignment.
asgn x not y goto N	x := not y;	Boolean NOT-assignment.
asgn x y op z goto N	x := y op z;	Assignment with operation *op*.
wait s goto N	wait(s);	Semaphore Wait() operation.
signal s goto N	signal(s);	Semaphore Signal() operation.
if x ifop y goto N else M	x ifop y	IF construction with *ifop* relation.

Note that each imperative instruction includes an obligatory goto clause, which states a line number of the next statement to be executed (in flowcharts we use arrows for the same purpose).

Each imperative intermediate language statement can also have an optional section "//description" (e.g. "asgn x y goto 5//x := y;"). State space diagram builder uses the first part of the statement (before "//") to analyze program behavior, and the second part (after "//") to print a current process line in state space diagram nodes.

Consider an example of intermediate language representation (Fig. 11, Ex. 8).

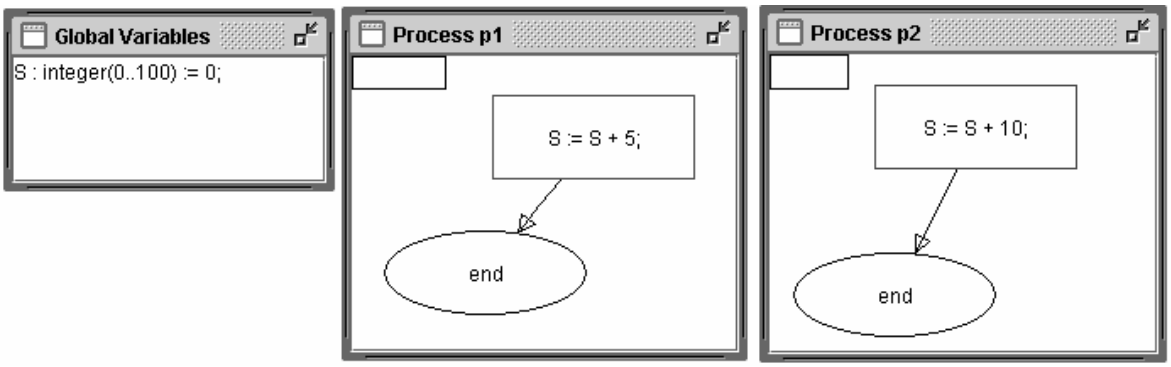


Fig. 11. Flowchart of a trivial concurrent program

Example 8. Intermediate language representation of a program from Fig. 11

```
__commonvariables
int S 0 100 0
__endofcommonvariables
__process p1
__code
asgn S S + 5 goto 1//S := S + 5;
__endproc
__process p2
__code
asgn S S + 10 goto 1//S := S + 10;
__endproc
```

4.3. Visualization Module

This module is intended for actual state space diagram building. You have two choices: either to press Initialize button to expand only starting state and then perform manual step-by-step expansion, or to press Expand All to build entire diagram at once. It is also possible firstly to expand several states by hand and then continue in automatic mode.

The visualization algorithm works in the following way. Firstly it generates starting state. It is quite simple: we know initial values of the variables, and the first line of each process becomes current in it.

To expand any existing state (i.e. to find all derived states), an algorithm should simulate one step of program execution, and it is possible, since the current state provides any necessary information about the situation at the moment. An OS scheduler can execute a statement from any process (CPV statements are considered as atomic), so we should consider all possibilities. In pseudo-code it looks like this:

```
for every process p in the program
    State s1 := CloneCurrentState();
    s1.MakeOneStepInProcess(p);
```

```
        s1.Draw();
    end
```

During simulation the only tricky moment arises when dealing with semaphores: from a record like <S=0, wait(S)> it is impossible to conclude is the process already blocked on the semaphore S, or it is just about to execute wait(S) statement. That's why we need to have an additional flag for any process (blocked / not blocked) and an additional syntax for representing this flag on state space diagram nodes. In CPV this difficulty is solved by means of using color: blocked processes are marked with red[7]. If two nodes contain identical information about variables values, but some process is marked with different colors, they are treated as different also. Another semaphore point concerns logic of statement execution. For any non-semaphore operation CPV executes something like

```
PerformOperation();
GotoNextFlowchartBlock();
```

In case of semaphore operations the logic is different:

```
PerformOperation();
if(ProcessIsNotBlocked())
    GotoNextFlowchartBlock();
```

Drawing a state is not so trivial also. For a start, we should remember all states, which are already on the screen in a separate set. If the state you are about to draw is already drawn, it is enough to make a transition from the parent state to it; otherwise we should firstly paint the new state. Transitions in CPV are different. Mostly they are straight, but if there is a transition from state A to state B and conversely, CPV changes them to curved arrows, see Fig. 12 (this concerns both visualizer and flowchart editor)

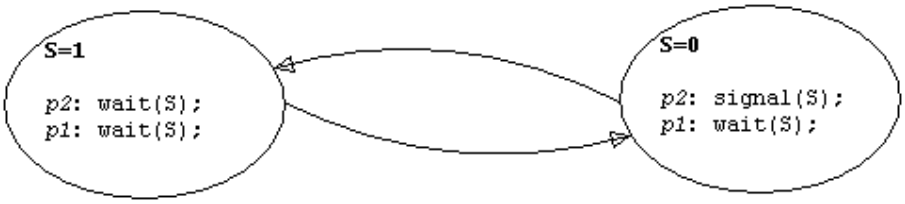


Fig. 12. Curved arrows in CPV

To perform automatic state expansion CPV executes the following code:

```
put all onscreen states into MySet set
while MySet is not empty
    s := MySet.TakeOutAnyElement();
    Draw s on the screen, if necessary
    if s is not analyzed yet
        generate all states, derived from s, then put them into MySet set
        mark s as analyzed
```

[7] CPV also uses blue font to mark the final state, but it is just a "color sugar".

```
        end
    end
```

This algorithm works until all states are drawn on the screen and have an attribute “analyzed”.

There is one more point, related to the state expansion algorithm. In real systems the difference between breadth-first and depth-first expansion is important (any good book on graphs should provide suitable cases for each of these approaches). In our case we expand all remaining states at once (when user presses Expand All button), so it doesn’t matter which algorithm to use. An actual solution is hidden inside an implementation of set type. In CPV I used HashSet class from the standard Java library as a type of MySet; therefore, I have no right to discuss its behavior.

Additional attention should be paid to graph layout manager: it is not enough to produce states; we should also place them on the graph surface in some reasonable way. For now a very simple scheme was applied (Fig. 13).

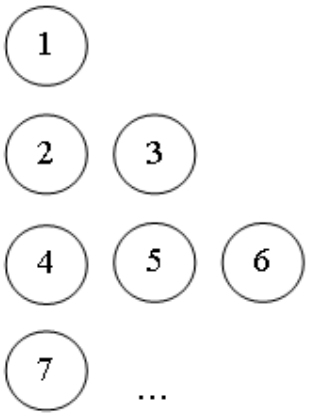


Fig. 13. CPV graph layouting scheme

Graph nodes are moveable, so it is possible to rearrange them manually. Examples of generated state space diagrams are shown on Fig. 14 and Fig. 15.

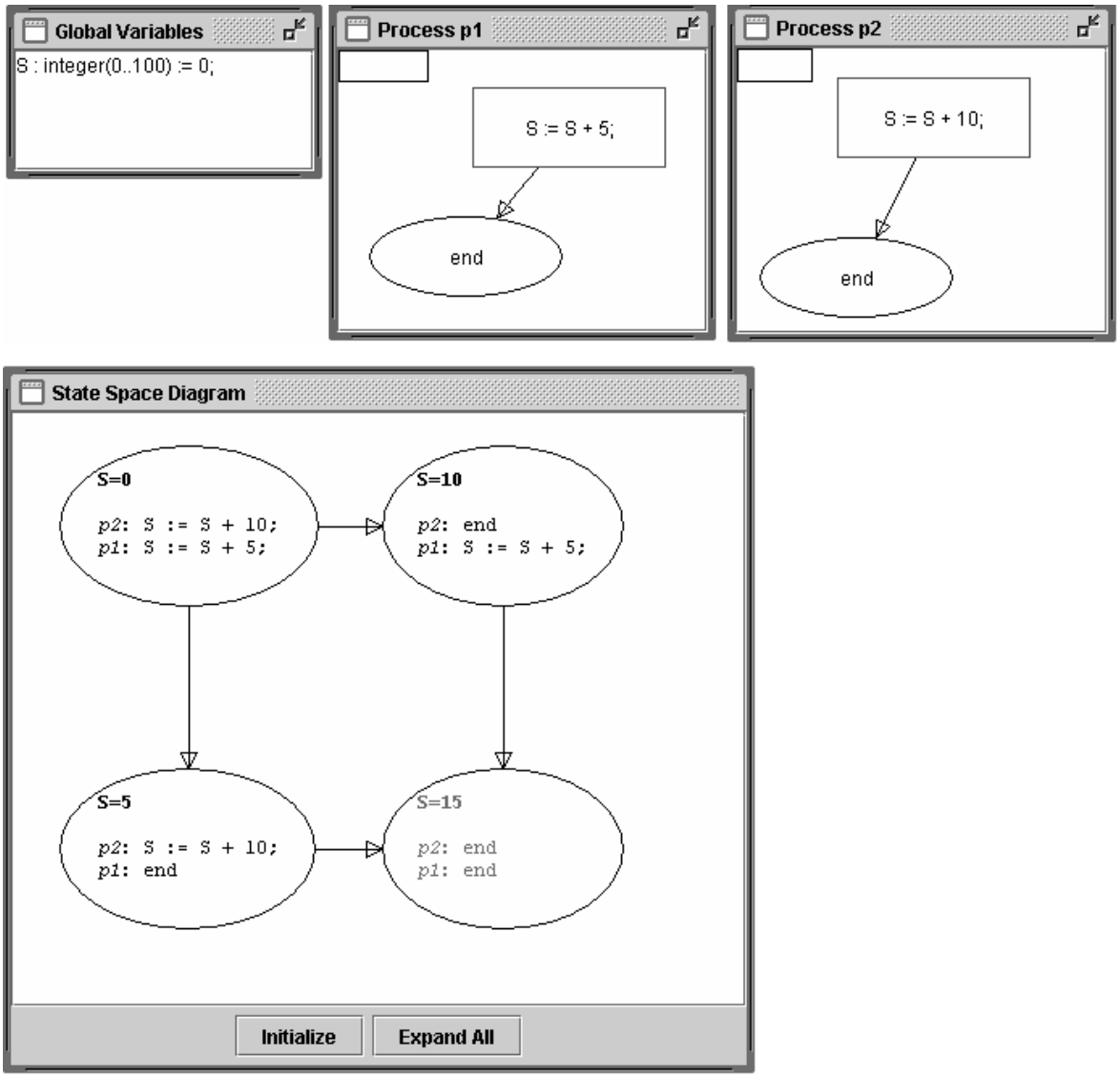
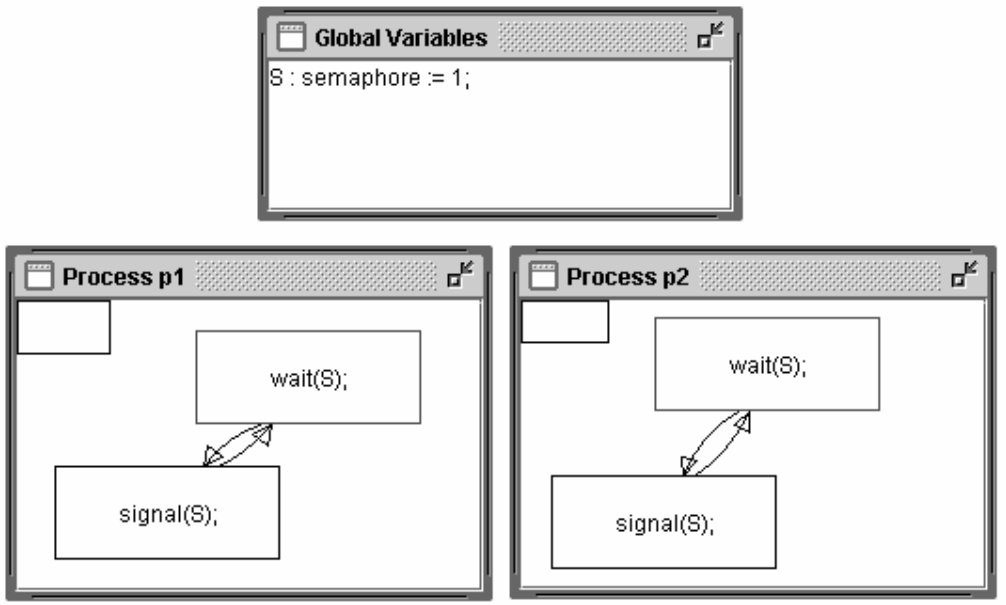


Fig. 14. State space diagram of a trivial concurrent program



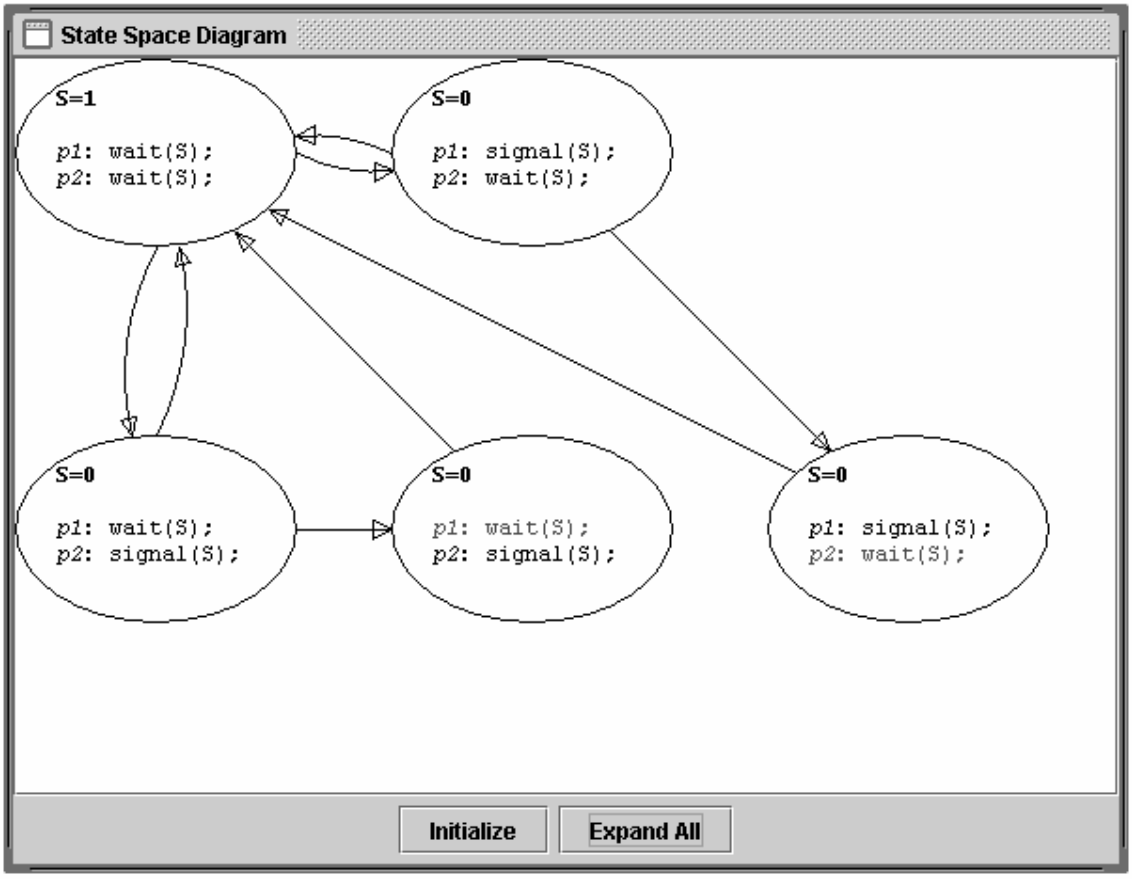


Fig. 15. State space diagram of a concurrent program with semaphores

4.4. JGraph

Since two CPV modules — the flowchart editor and the visualizer — implement, basically, a bunch of graph-drawing functions, it is reasonable to have some generic framework. For this purpose CPV uses an open source JGraph library. JGraph is quite big (dozens of classes) and complicated extensible Swing component for graphs visualization. JGraph itself provides only quite basic possibilities for graph handling: you can create simple labeled rectangular nodes and connect them using straight arrows. On the other hand, it is possible to provide custom node/vertex views, mouse handlers, etc., so the flexibility of JGraph is very high; for now it satisfies all our requirements. CPV seriously extends JGraph classes (Fig. 16).

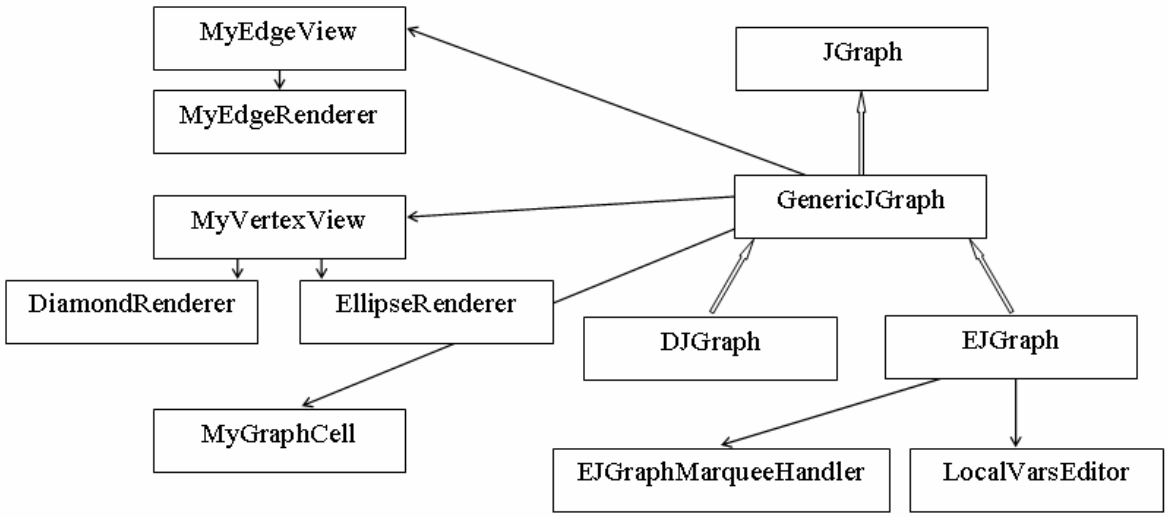


Fig. 16. CPV JGraph extension

Although JGraph is really powerful and extensible, I should say it is not so *easily* extensible and easy to use in general. To my mind, JGraph architecture is very complicated and not always rational. In many occasions JGraph requires you to spend a lot of time to make rather simple things; on the other hand, it can be a very good training for your nerves.

It stands to reason that JGraph is not just the first graph visualization library I found inside the Web. It is worth to mention other possible alternatives.

The first one — to write our own library — we tried to avoid by all means. The concept of graph is very common in mathematics and computer science, so it is hard to believe that nobody created an acceptable solution before us.

There were several criteria for choosing the library among existing ones:

1. It should be written in Java, since we use Java.
2. It should be free (for the obvious reason) and open source, if possible.
3. It should be standalone and lightweight. I see no reasons to create a 5 Mb “free addition” to 100 Kb software package, even if this addition contains 1000 great, but useless (for us) classes.

JGraph author, Gaudenz Alder, position his work as “the most powerful, lightweight, feature-rich, and thoroughly documented open-source graph component available for Java”. Although this characterization looks very ambitious, I have to agree with, at least, most its aspects.

JGraph lacks several important features (in my opition), and the documentation is far from ideal also, but I was unable to find better library.

Most good graph visualization packages, such as MonarchGraph or yFiles, are commercial, and the common problem of almost all free libraries is the lack of informational resources. Programmers often consider the process of writing technical documentation as a very boring activity, and, therefore, many good libraries remain obscured. The sites of such projects frequently consist of one or two pages with a short description, optional screenshots and only one link, proudly entitled “downloads”. Libraries like Graph Visualization Framework or Otter remained out of our scope due to these problems.

The truth becomes known only in comparison. And after consideration of several libraries, it is clear, that the documentation of JGraph is not so poor, the number of included features is more than just satisfiable and so on. You also have an access to a good quickstart tutorial, FAQ, detailed JavaDoc documentation, several ready-to-use examples and a serious Internet forum.

CPV Links:

CPV homepage: http://stwww.weizmann.ac.il/G-CS/BENARI/cpv/

JGraph homepage: www.jgraph.org

5. Using CPV in Teaching

CPV is intended to be an educational tool. Now we'll discuss situations, where CPV can be utilized. As I mentioned before, the original idea was to provide a good tool both for students and teachers. To provide a more or less sensible basis for the conclusions it was decided to arrange a small experiment on CPV usability.

5.1. Preliminaries

Before performing any actions, it is necessary to formulate our aims more precisely:

1. How a student can use CPV when studying concurrency? Why he/she should use CPV?
2. How a teacher can use CPV in his/her work? Why he/she should use CPV?

In other words: a teacher/student can use either traditional pencil and paper or CPV. What advantages can CPV usage bring? Any teacher looks for some "visual" methods, which can simplify teaching process, can help teacher in explanation of the material. Is CPV a kind of such helpful tool, which can make a process of teaching simpler and more effective (for a teacher)?

Students look for a tool, which can help them to understand the material better in a smaller amount of time. Is CPV a kind of tool, which provides better, more fundamental understanding in a short space of time (in comparison with traditional approaches)?

What points of teaching/studying CPV actually affects?

5.2. Methodology

Firstly I've gathered a group of mostly second-year students (about ten persons), not familiar with concurrent programming concepts. After that I've conducted a tutorial on concurrency. Both slides and CPV were used. Then these students were asked to solve several problems. The problems were designed to be easily solvable via our tool, but nobody enforced students to use CPV instead of pencil and paper.

5.3. Teaching Experience

Here I'd like to explicitly mention ways of CPV usage during tutorial session (i.e. CPV as a *demonstrational* tool):

1. Explaining the concept of state space diagram. Very simple, but good way of utilizing CPV. By means of trivial examples, it is possible to demonstrate numerous generated state space diagrams, to show different ways of execution for multithreaded programs and list-like diagrams for single-processed applications.
2. Explaining potential problems, which can be caused by the existance of various execution scenarios. As I mentioned before, generally we should be able to obtain the same results regardless of actual execution flow, which means (for most simple educational programs, which can be handled by CPV) the uniqueness of the final state. A good example of a program with unique final state is shown on the Fig. 2; the Fig. 17 demonstrates a program with multiple final states.

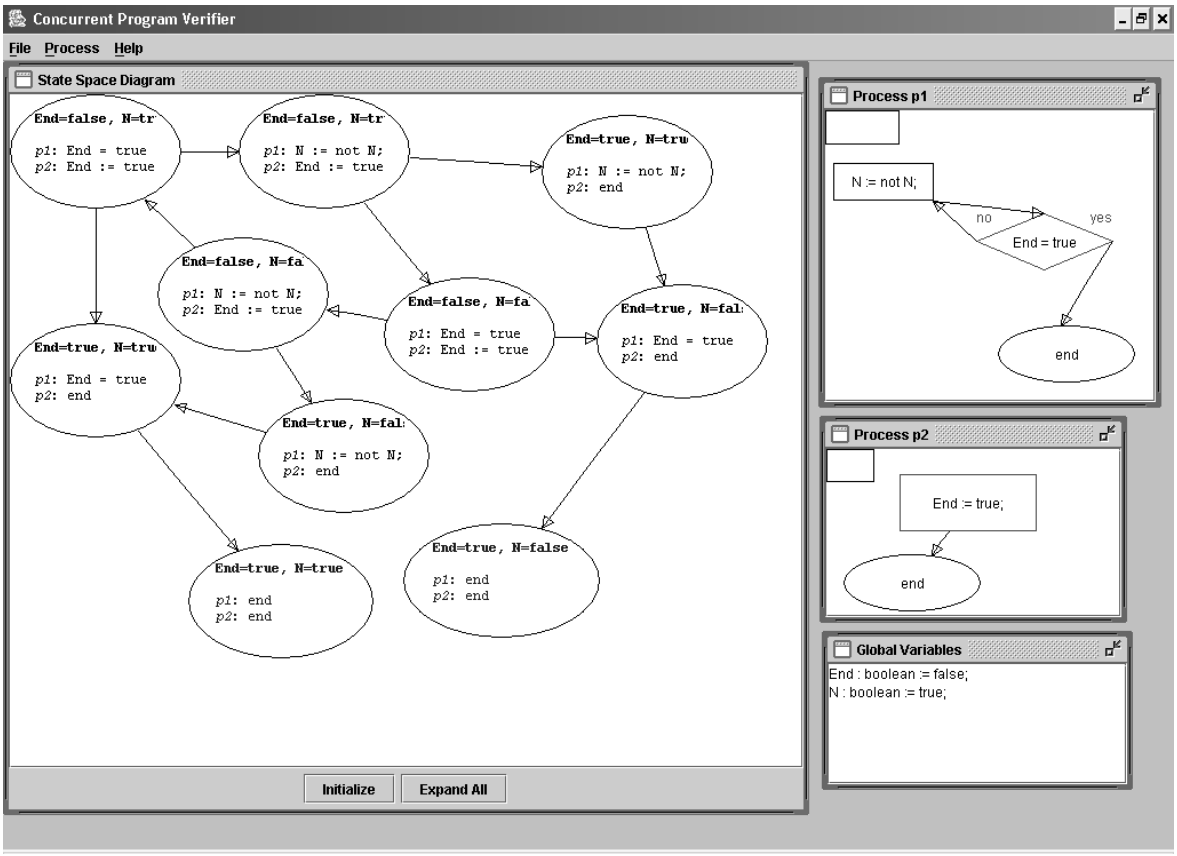


Fig. 17. A program with multiple final states

3. Semaphores as mutexes. The concepts of mutual exclusion, critical sections and deadlocks. Semaphores in CPV were considered earlier; here I'll mention that state space diagram demonstrate deadlocks in a very clear manner (Fig. 18; both processes are blocked).

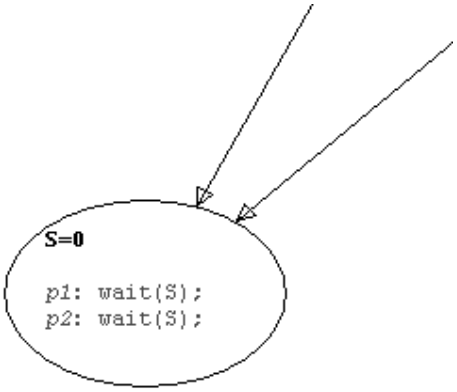


Fig. 18. Deadlock example in CPV

4. Semaphore as a synchronization mechanism. Semaphores can be also used to specify that a certain code should be executed only after some other actions. For instance, consider the following concurrent array sorting routine:

```
GLOBAL S : Semaphore := 1;
```

```
process1: SortFirstHalf();
          Signal(S);

process2: SortSecondHalf();
          Signal(S);

process3: Wait(S);
          Wait(S);
          MergeHalves();
```

Here a semaphore doesn't allow MergeHalves() execution until both array halves are sorted.

5. Basic concepts of program verification. As it said before, during the verification process we usually have to check some properties of the program. There are two kinds of such properties: *savety properties* and *liveness properties*. Informally speaking, safety property means that bad things never happen while liveness property ensures us that good things eventually take place. Have a look, for example, at the simple program with semaphores, shown on the Fig. 15. For this program we can formulate two safety properties (there is no deadlock scenario; at most one process enters the critical section) and one liveness property (a process, trying to enter the critical section, must eventually succeed in doing so). With CPV we can check fulfilment of such properties by examination of the state space diagram. "Bad things never happen" (safety) means the absence of "bad" states. Thus, "no deadlock scenario" means the absence of deadlock states (see Fig. 18), and mutual exclusion property can be proved if we found no states, where both processes are in their critical sections. To prove liveness properties we should analyze all possible execution scenarios and check if they all lead to the desired result. A good example program for checking liveness properties is Ex. 2, Fig. 2. Here two liveness properties should be satisfied: eventually i=3, j=3; eventually the program should terminate. The analysis of all execution scenarios shows: every path leads to the unique final state, where i=3 and j=3, so our properties are fulfilled.

Surely, my own opinion about CPV cannot be objective, since I am also a developer of this tool. On the other hand, not only my observations make up a basis for making conclusions. I also discussed CPV with other teachers and people, whom I consider as experienced users.

Overall conclusion is simple: CPV usage during the lectures is defensible. Our software will not substitute slides, but can serve as a kind of auxiliary tool for demonstrating the concepts on the middle level (lectures are to "high-leveled" and far from the reality, while actual programming language code is too technical, "low-leveled" for the first consideration of concurrency). The experience of applying CPV in practice showed several directions for the possible future improvement. The most serious observation concerns transitions visualization. When you see a state with, say, three outgoing arrows, it is not so easy to figure out the relation between them and corresponding processes. A possible improvement suggests using different colors for different processes/transitions. Suppose that the process *p1* is marked with brown. Then if you want to know what happened after one step in process *p1*, just study brown outgoing arrow.

Another idea is related more to CPV package than to CPV itself. It was noted that any good educational software package includes standard examples, which can be directly utilized by

the teacher. Thus, it is reasonable to add ready-made flowcharts for deadlock explanation, Dekker's algorithm, etc. to CPV package. A kind of "quickstart guide" was considered as a good addition also.

5.4. CPV for the Student

A list below contains all suggested problems during the experiment.

1. Prove mutual access exclusion property. Find the possibility of deadlock.

```
GLOBAL int Turn := 1;


proc1: for(;;)
       {
           NON_CRITICAL_SECTION;
           while(Turn != 1)
               ;
           CRITICAL_SECTION;
           Turn := 2;
       }

proc2: for(;;)
       {
           NON_CRITICAL_SECTION;
           while(Turn != 2)
               ;
           CRITICAL_SECTION;
           Turn := 1;
       }
```

2. Find a deadlock scenario.

```
GLOBAL semaphore S := 1;
GLOBAL Boolean B := true;


proc1: Wait(S);
       if(B == true)
       {
           B := false;
           Signal(S);
       }
```

```
        Wait(S);
        B := true;
        Signal(S);

proc2: { THE SAME }
```

3. Prove/disprove the uniqueness of the final state in the program.

```
GLOBAL Boolean N := false;
GLOBAL Boolean B := true;

proc1: while(B)
           N := not(N);

proc2: while(B)
           if(N == false)
               B := false;
```

The students had about 1.5 hours to complete these tasks, and then I talked with each of them personally about CPV.

The biggest surprise for me was the fact that many students regarded these tasks as pretty hard; maybe I just forgot myself at their age and educational level. Only several students showed really deep understanding of the material. They correctly solved proposed problems and presented CPV screenshots with solutions. Despite these difficulties with problem solving, I've received some valuable information from the group.

Although politeness is a good thing in general, sometimes people are too polite. Most students tried to avoid direct CPV criticizing. It was quite typical to receive a response like: "your tool is good, but it can be even better if you do X". One interesting idea was to add "statistics": the student thinks CPV should output some information about the program, such as the number of states and final states, the number of deadlocks, etc.

Among positive replies students note good CPV demonstrative and usability sides: "It makes the diagrams easier to understand", "It made easier to understand what is it all about and how program really works", "I don't know if there are other (possibly better) tools for demonstrating problems one might encounter using multiple threads, but CPV worked reasonably well", "It is quite easy to use".

Negative responses mostly concern minor problems in user interface, which will be fixed in the nearest future. Some students also mentioned a well-known problem: for any "real-world" problem state space diagram becomes too large for manual exploration.

6. Some Conclusions

Although it is very simple, CPV has the potential to be a useful tool for teaching an introduction to concurrent programming. For now, CPV can be used to demonstrate basic concepts of multithreaded programming by means of small examples. It can be utilized also by students as an “educative toy” suitable for self-studying matters.

To the present moment, only 1.0 version was released. Now we have several good ideas for the further development and evaluation, which will be eventually realized. Here is a short summary:

1. Wider color usage. Marking different processes and outgoing edges with different colors. This change should noticeably increase the “visuality” of our software, since it will be much easier to consider various execution scenarios.
2. More complete software package. A good tool should include a quickstart guide and ready-made examples, suitable both for a student and for a teacher (as “standard” addition to the lecture notes).
3. Various user interface improvements. From the very first implementation (“alpha version”) CPV interface noticeably evolved. For instance, we added templates for different commands and variable declarations, rewrote arrow creation method, added support for multi-segmented arrows. But there are still enough places for improvements. For example, there is a contradiction: if state space diagram nodes are small, you can see bigger portion of the graph on the screen, but only a part of each state’s textual representation and vice versa. One of ideas is to implement single state zooming to enable user easily magnify single state without a need to resize them.
4. Possible integration with Spin. I’ve already mentioned this model checking tool. Now this direction of our work is on a very early stage, so it is quite untimely to discuss its aspects, but the basic idea is to somehow combine CPV visualization capabilities with Spin model checking functions to obtain a much more powerful educational instrument.

7. References

[AGH00] K. Arnold, J. Gosling, D. Holmes: The Java Programming Language, Third Edition. Addison-Wesley, 2000

[BBCCKMSU96] N. Bjorner, A. Browne, E. Chang, M. Colon, A. Kapur, Z. Manna, H. Sipma, T. Uribe: STeP: Deductive-Algorithmic Verification of Reactive and Real-time Systems. International Conference on Computer Aided Verification, pp.415-418. vol. 1102 of Lecture Notes in Computer Science, Springer-Verlag, 1996

[Ben-Ari00] M. Ben-Ari: The Bug That Destroyed a Rocket, 2000

[Ben-Ari90] M. Ben-Ari: Principles of Concurrent and Distributed Programming. Prentice Hall, Cambridge, 1990

[Ben-Ari98] M. Ben-Ari: Constructivism in Computer Science Education. The Proceedings of the 29th ACM SIGCSE Technical Symposium on Computer Science Education, Atlanta Georgia February 25th – March 1st 1998, pages 257 – 261

[BL96] J. Bengtsson, F. Larsson: Uppaal — a Tool for Automatic Verification of Real-Time Systems. DoCS Technical Report Nr 96/67, Uppsala University, 1996

[BSRP97] J. Bergin, M. Stehlik, J. Roberts, R. Pattis: Karel++. A Gentle Introduction to the Art of Object-Oriented Programming. John Wiley & Sons, 1997

[Dijkstra68] E. Dijkstra: Cooperating sequential processes. Programming Languages Academic Press, New York, 1968

[Feldman96] M. Feldman: Software Construction and Data Structures with Ada 95. Addison-Wesley, 1996

[Hansen02] P. Brinch Hansen: The Origins of Concurrent Programming: From Semaphores to Remote Procedure Calls. Springer Verlag, 2002

[Holzmann03] G. Holzmann: The Spin Model Checker. Primer and Reference Manual. Addison-Wesley, 2003

[KP99] B. Kernighan, R. Pike: The Practice of Programming. Addison-Wesley, 1999

[Lea99] D. Lea: Concurrent Programming in Java: Design Principles and Pattern (2nd Edition). Addison-Wesley, 1999

[MK99] J. Magee, J. Kramer: Concurrency: State Models & Java Programs. John Wiley & Sons, 1999

[Persky99] Y. Persky: SimAda Concurrency Simulator. MSc thesis at Tel-Aviv University, 1999

[Snow92] C. Snow: Concurrent Programming. Cambridge University Press, 1992

[TDT97] S. Taft, R. Duff, T. Taft: Ada 95 Reference Manual: Language and Standard Libraries: International Standard Iso/Iec 8652:1995(E) (Lecture Notes in Computer Science, 1246). Springer Verlag, 1997

[Whiddet87] D. Whiddet: Concurrent Programming for Software Engineers. Ellis Horwood Ltd, Chichester, England, 1987

[Yoder90] S. Yoder: Introduction to Programming in Logo Using Logo Plus. Intl Soc for Tech in Educ, 1990