Concurrent Program Verifier

by Maxim Mozgovoy and Mordechai (Moti) Ben-Ari

USER GUIDE

The Purpose of the System

As it can be understood from the project title, the purpose of this software is verifying various concurrent programs. Designing and debugging an ordinary (single-processed) program is always non-trivial task, and it becomes even more complicated when we are working with concurrent programs. Concurrent Program Verifier can help you to understand the nature of concurrent programming, to realize concurrent behavior, and, possibly, to avoid some common obstacles in real projects. Concurrent Program Verifier (CPV) is not an enterprise-level system; it is designed for educational purposes only. CPV allows you to perform two basic tasks:

- to develop your own concurrent program using built-in flowchart editor;
- to analyze its behavior by means of special state space diagram builder.

We use flowcharts to build concurrent programs instead of some ordinary text-based programming language. Our flowchart editor gives an ability to formulate basic statements, such as assignments, conditionals and semaphore operations (this seems to be enough for simple educational programs).

State space diagram is a standard way to describe a behavior of a program (both single-threaded and concurrent). Having such diagram, you can easily analyze possible workflows and test your program for correctness.

Interface Overview

During the process of editing/analyzing a concurrent program, you can see several windows (fig. 1):

- a global variables frame;
- a separate frame for each program’s process;
- a state space diagram frame.

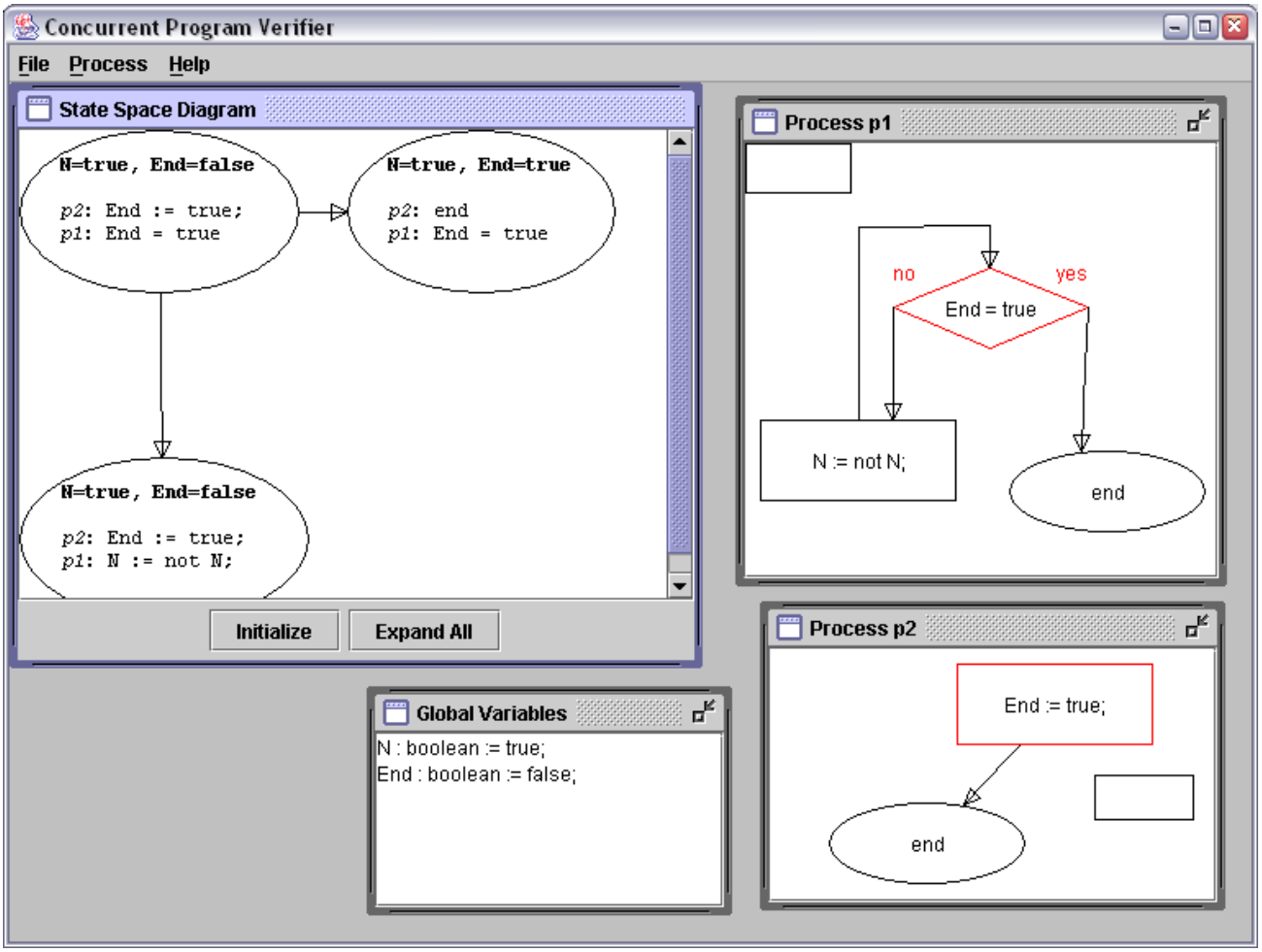

Fig. 1. Main application window

The global variables frame is designed to hold all global (common) variables of your concurrent program. Each process frame is a drawing surface: here you can create a flowchart, describing the corresponding process. You will have one such frame per process, so the number of process frames always equals to the number of processes in the program. The state space diagram frame is a placeholder for the state space diagram.

Basic project management functions are available through the menus. Via menu **File** you can create, load, save and close projects; menu **Process** allows you to add, remove and rename single processes; menu **Help** just shows an About box.

Programming in CPV

Declaring Global Variables

CPV uses Pascal-like syntax for variable declaration. There are only three types of variables: integers, Booleans and semaphores:

Syntax	Example
VarName : integer(MinValue..MaxValue) := InitValue;	a : integer(-5..100) := 11;
VarName : boolean := InitValue;	flag : boolean := true;

VarName : semaphore := InitValue;	S : semaphore := 3;

An integer variable is, basically, an ordinary integer, similar to the variables of such kind in any programming language. Note, although, that in CPV any integer should be **always** bounded (i.e. you have to supply lower and upper bounds for its value).

A Boolean variable is also an ordinary instance of a standard logical data type; its value can be either **true** or **false**.

A semaphore (Dijkstra's term) is a special variable, used to control mutual access to the shared resources (e.g. global integer variables). Actually, a semaphore has an integer value domain, so its initial value is an integer.

As it was said before, you should put all your global declarations into the global variables frame. Only one declaration per line is allowed.

You can use "fast templates" to speedup declaration process. Right-click somewhere in the global variables frame to invoke a popup menu containing three patterns for three different variable types. Select desired one to insert it immediately into the global variables frame.

Programming Processes

Adding a new process is simple: select **Process**, **New** menu option and type the name of the new process.

When a process is created, a Local Variables Editor Frame (fig. 2) is being invoked. This frame works exactly as the Global Variables Frame, but it is intended for local variables (inner variables of a process) declaration.

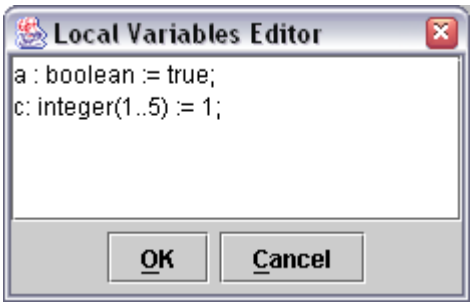

Fig. 2. Local Variables Editor Frame

After pressing **OK** or **Cancel** in the editor box, you can begin working with the just created process window (fig. 3).

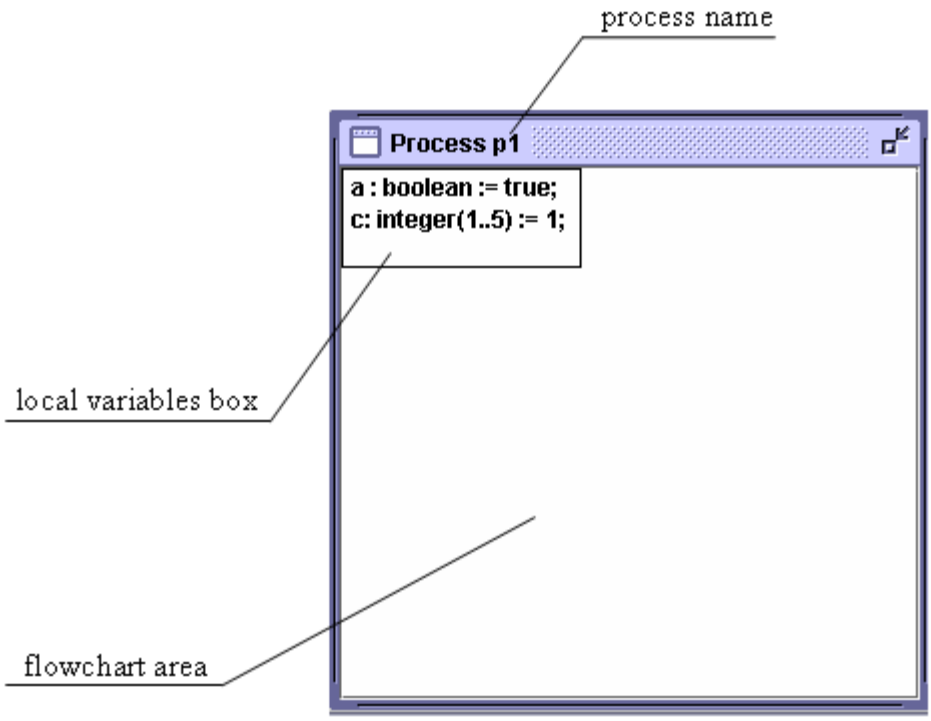

Fig. 3. Process window

Now let's consider some techniques of the flowchart building process. Basically, a flowchart consists of blocks and arrows, so you have to know how to manage blocks and create/destroy arrows between them.

There are four block types: assignment, semaphore operation, branching and end block. To create any block, right-click somewhere in the process window and select desired block type using pop-up menu. When creating an ordinary (non-end block), an input dialog box is being invoked; you can use it to edit block data.

Assignment blocks can hold various assignment operations:

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;	address := 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;	number := number – 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.

Semaphore operation blocks are designed to contain one of two semaphore operations:

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

Branching blocks are used for describing conditions:

Syntax	Examples	Comment
value1 = value2	a = 5 height = width flag = true	Equality test: value1 and value2 are variables or constants of the same type (integer or Boolean).
value1 <> value2	a <> 5 height <> width flag <> true	Inequality test: value1 and value2 are variables or constants of the same type (integer or Boolean).
value1 < value2	a < 5 a < b	"Less than" relation test: value1 and value2 are integer variables or constants.
value1 <= value2	a <= 5 a <= b	"Less or equal" relation test: value1 and value2 are integer variables or constants.
value1 > value2	a > 5 a > b	"Greater than" relation test: value1 and value2 are integer variables or constants.
value1 >= value2	a >= 5	"Greater or equal" relation test: value1 and

	a >= b	value2 are integer variables or constants.

The purpose of end block is an end-of-process indication. You can have only one end block in each process (or no end blocks at all), so if there are several blocks, which lead to the end-of-process, you should bind several arrows to one end block instead of creating multiple end blocks (fig. 4, down instead of fig. 4, up).

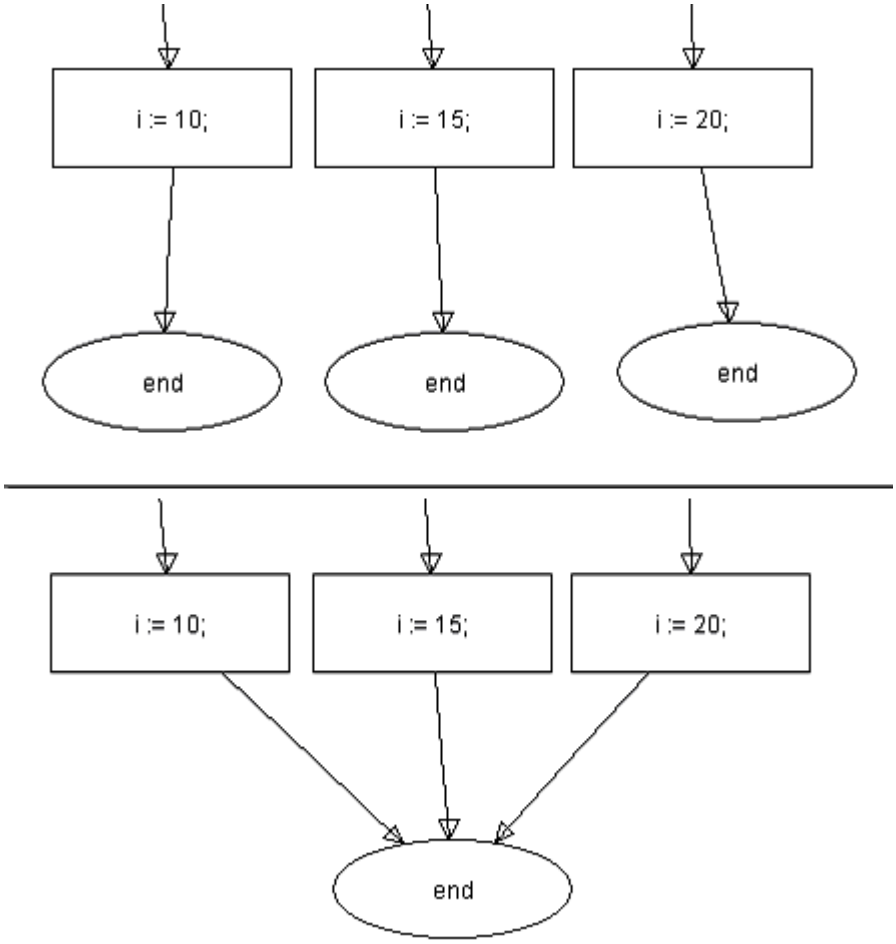

Fig. 4. Placing end block

Double click on any block (including Local Variables Frame, excluding end block) will popup a data editor window for it.

You can also move and resize blocks in usual way (using the mouse).

The next step is arrow creation. Each block (excluding Local Variables Frame) has one or more "snap points" (fig. 5).

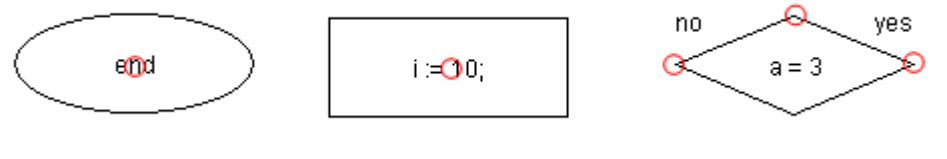

Fig. 5. Snap points of various block types

You can connect any pair of snap points, belonging to different blocks, with an arrow. To do it, move a cursor to the neighborhood of the source snap point (cursor will be changed to “hand”), hold left mouse button, move a cursor to the neighborhood of the target snap point and release the button. It is possible to create multi-chained, bendable arrows (they have the same functionality, but in some cases look better). Left-click on any arrow while holding **Shift** key to add/remove a moveable point (Fig. 6).

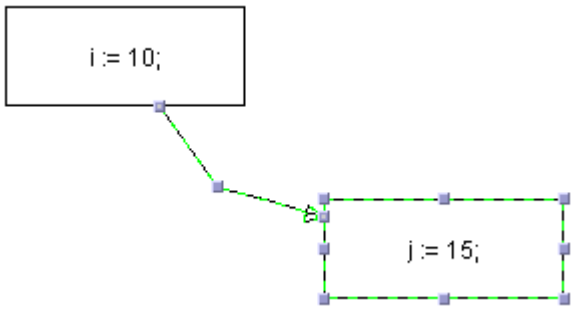

Fig. 6. Moveable arrow points

Select any onscreen object (except Local Variables Frame) and press **Del** key to remove it from the flowchart.

Exactly one block in each process should be marked as starting. Right-click on the desired block and select **Mark As Starting** option.

Understanding State Space Diagrams

State space diagram building is a useful technique for analysis and verification of concurrent programs. Since our computers are finite machines, any program (either single-threaded or concurrent) basically performs some transitions between available states of the computer (it’s what is actually done by a program). In CPV we are dealing with three kinds of values:

- global variables;
- local variables;
- the number of current instruction in each process.

Therefore, each state of our computer can be **uniquely** described by a vector, which contains current values of these parameters. In real computers there are some auxiliary parameters, having influence on the current state of the computer, such as other programs (when running a multitasking environment), interrupts, generated by a hardware, etc. Although, in many cases when you are interested in the behavior of your program only, they are not valuable.

Consider a simple single-threaded program (Fig. 7).

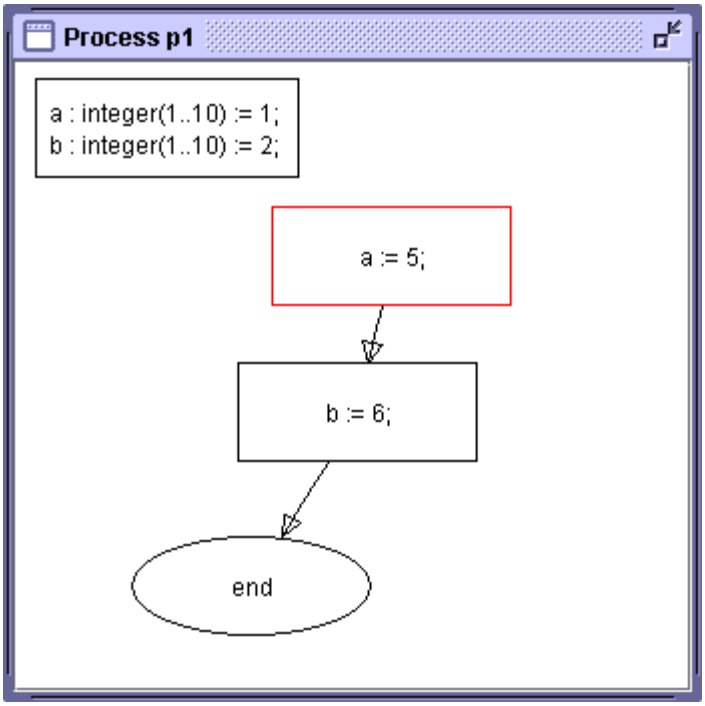

Fig. 7. Sample single-threaded program

Here we have no global variables at all, so its behavior can be described with a graph, holding current values of the local variables and the number of the current instruction in its nodes (Fig. 8).

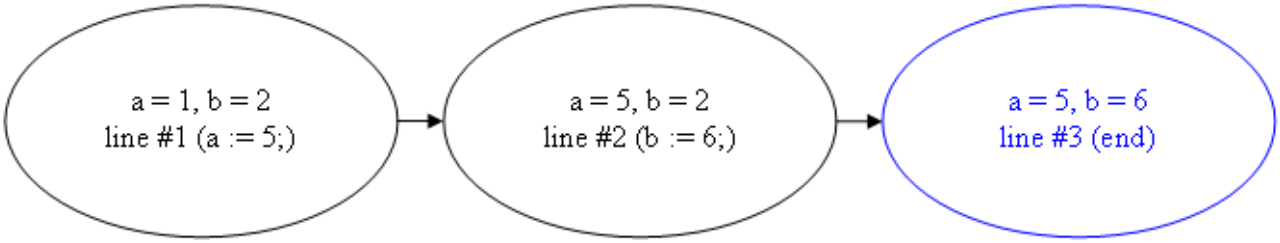

Fig. 8. State space diagram of the sample single-threaded program

You can see that the graph has a very simple, linked list form. Any single-threaded program produces a graph of such kind, so it is possible to (more or less easily) build the state space diagram by hand; in case of concurrent program the situation becomes more complex.

Consider a two-processed program (Fig. 9).

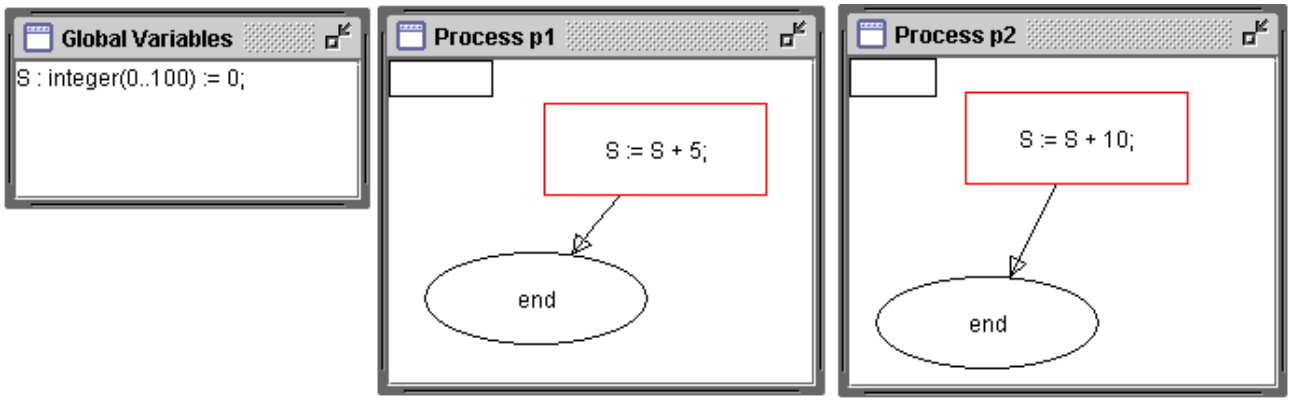

Fig. 9. Two-processed sample program

Now we have one global variable (S), but no local ones. Let's build a state space diagram for this program (Fig. 10; line numbers are omitted).

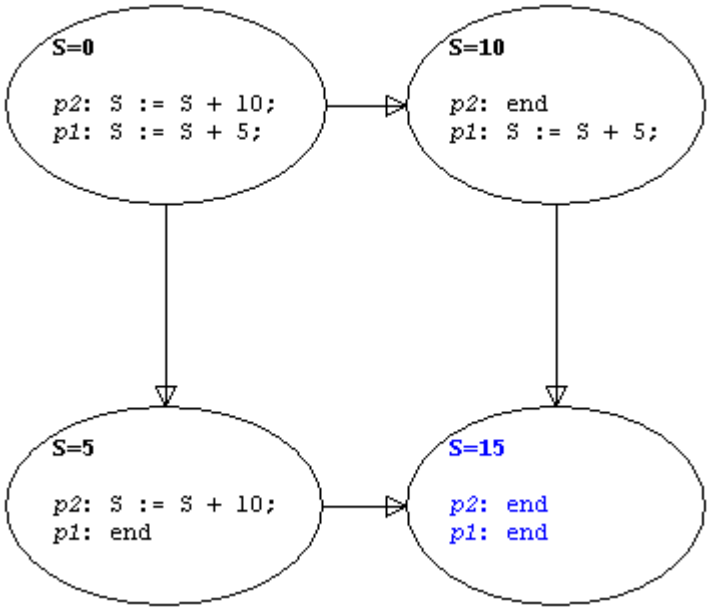

Fig. 10. State space diagram of the sample two-processed program

As it can be seen from the graph, a processor should every time decide which process' instruction will be executed on the next step. In the very start we have no idea, which instruction will be executed next (it is either "S := S + 10;" or "S := S + 5;"), so we should consider both possibilities. Only one process remains active during the next step, therefore the last transition is obvious.

Sample concurrent program is correct, since the result of its execution doesn't depend on an actual workflow (i.e. on the way of selecting the next instruction to be executed). If different workflows lead to different results, it is generally considered as a very bad situation; such programs are incorrect.

CPV can build such diagrams automatically via State Space Diagram window (Fig. 11).

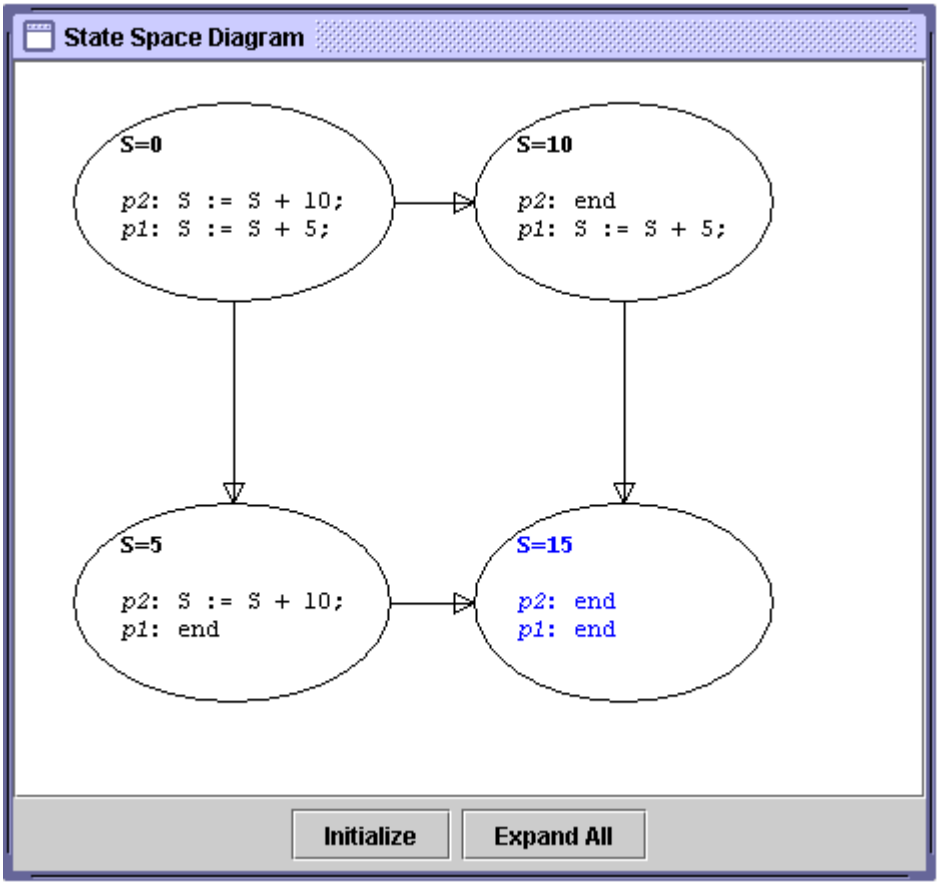

Fig. 11. State Space Diagram window

At first, you should press **Initialize** button to generate starting state of the program. Then you can double-click on any generated state to expand it (i.e. to build all derived states). Alternatively you can press **Expand All** button to expand all remaining states at once (and obtain a full diagram).

All graph nodes are moveable and resizable.

State Space Diagrams and Semaphore Operations

Semaphore operations bring some unusual behavior to the program, since individual processes can be **blocked** (frozen) and **unblocked**. Recall the semantics of semaphore operations Wait() and Signal(). When the value of the semaphore S is zero, and some process attempts to execute an instruction wait(S), it will be blocked until some other process executes an instruction signal(S).

This fact (process blocking/unblocking) should be somehow shown on the state space diagram. Let's consider a basic program with semaphore operations (Fig. 12).

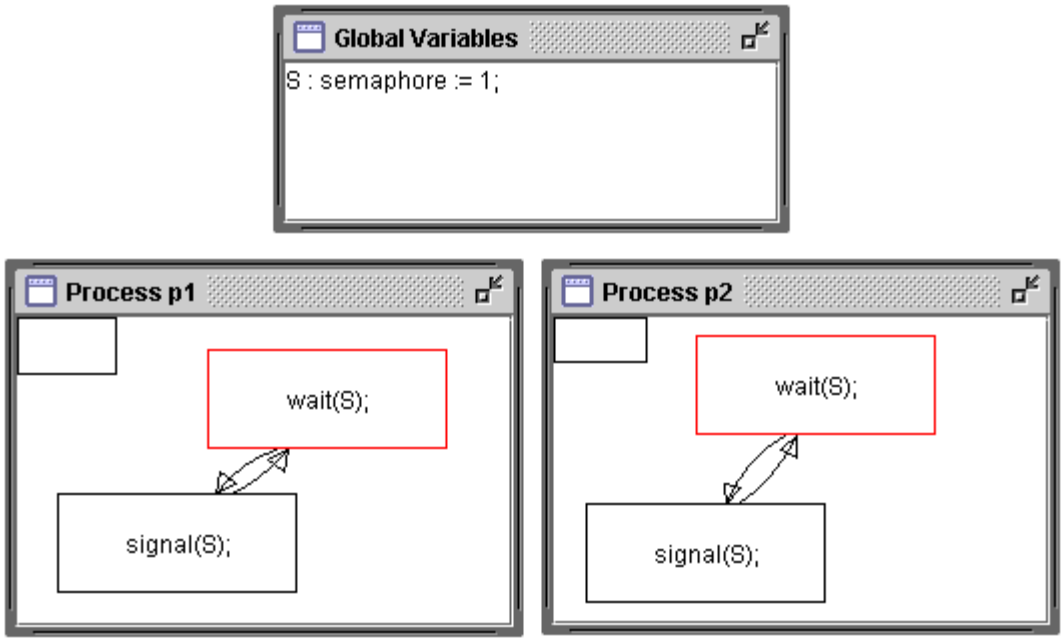

Fig. 12. Sample program with semaphore operations

Ready state space diagram for this program (obtained by pressing **Expand All** button in the State Space Diagram window) is shown on the Fig. 13.

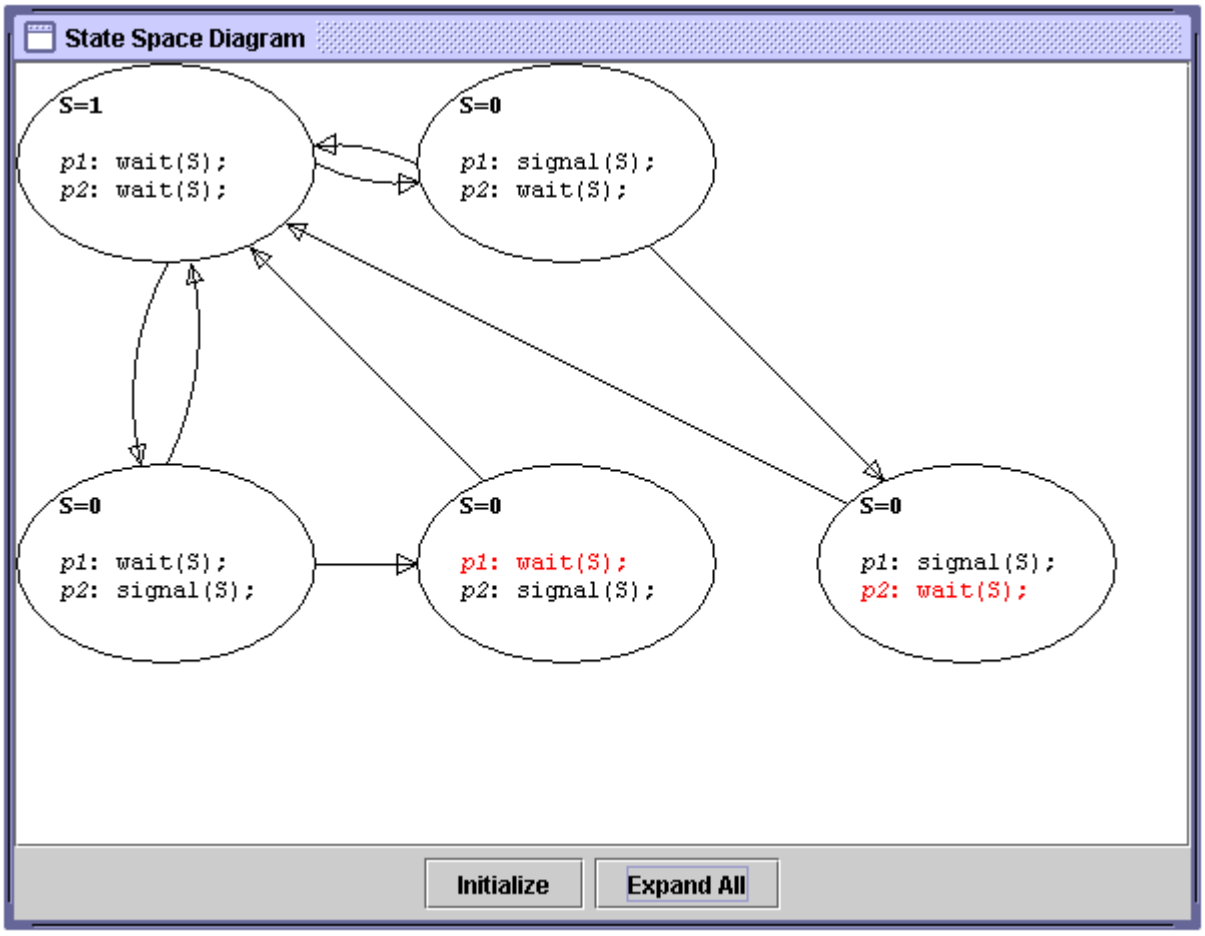

Fig. 13. Sample program with semaphore operations

As it can be seen, the described above situation (attempting to execute wait(S) while S=0) leads to blocking of some process. This blocked process is highlighted with red color.