Asynchronous Circuit and Systems Design Practical

August 12, 2011

1 Introduction

In this tutorial you will be working with *Workcraft*. It is a computer aided design (CAD) tool allowing to create and interactively simulate Petri Nets, Signal Transition Graphs, Circuits and some other model types. Beware, this tool is under heavy development and many basic features (such as Copy/Paste or Undo) are still not implemented.

The interface of the program is shown in Figure 1.

- *Main menu* is used to manage models, configure the system and call the external tools to do the additional processing;
- *Editor tabs* show all of the opened models;
- *Editor window* allows to view and edit a model;
- *Tool controls* is a special window which is active during the simulation. It allows to form new or follow the existing simulation trace;
- *Property editor* is used to edit properties of an object selected in the editor window;
- *Editor tools* panel allows to select mode of operation (as shown in Figure 1: Selection tool, Connection tool,). It will vary from one model to another providing different types of objects that can be created.
- *Workspace* presents the list of opened or imported files;
- *Utility windows* shows additional information such as external tool output, error messages or the progress of launched tasks.

The basic editor controls are as follows:

- Mouse wheel - zoom in and zoom out model;
- Left click - selects/connects objects, creates new objects (depending on the selected editor tool);

Right click - shows context-sensitive drop-down menu. (Can be used to create components with multiple outputs);

Middle button - used to pan view of the model.

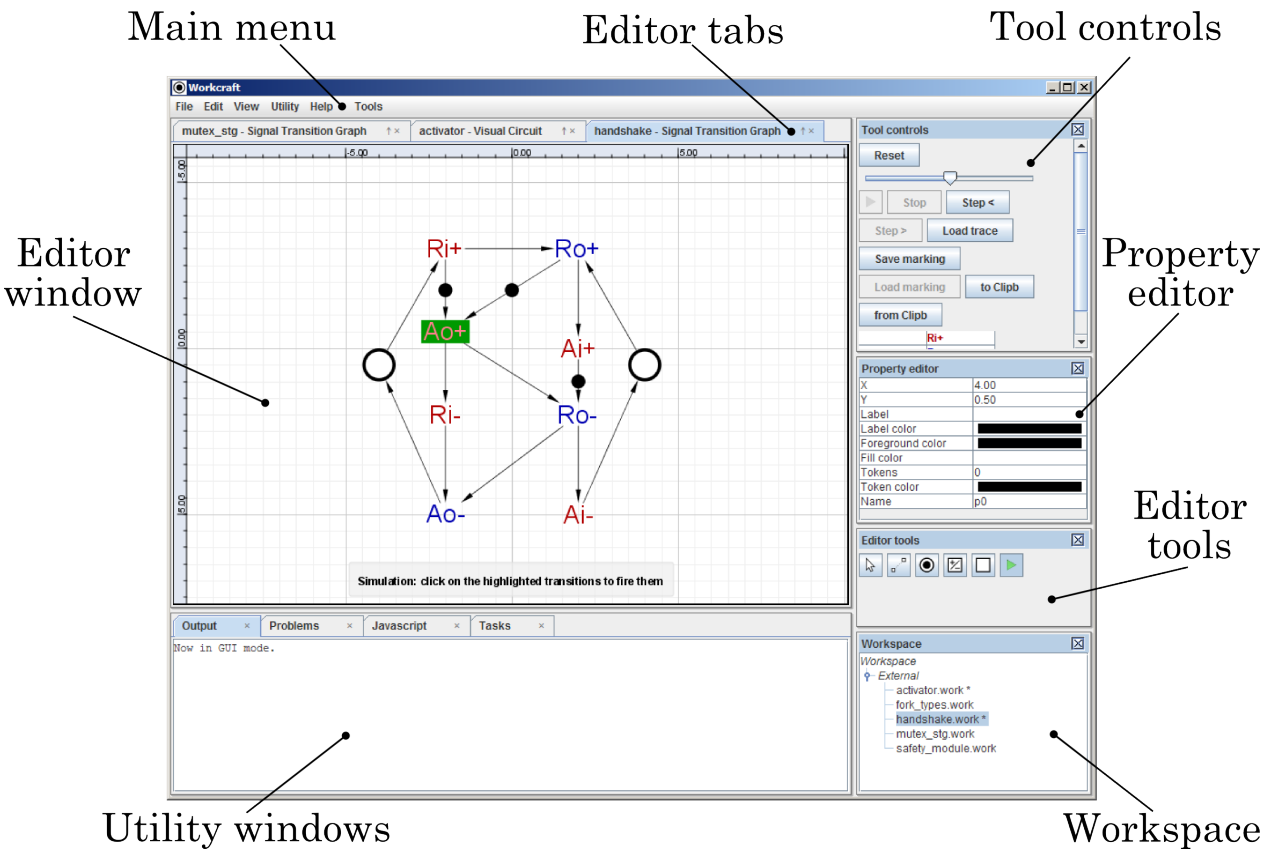


Figure 1: Workcraft interface

1.1 Basic editor modes (see editor tools)

Select - allows to select and move around various objects, change object properties or delete them.

Connect - create new directed connections between components;

Simulate - activate interactive simulation.

1.2 Additional editor modes for STG models

Dummy transition - creates a dummy transition;

Signal transition - creates a signal transition (a transition associated with a raising or falling edge of some signal). Use the selection mode to change its name or direction (input, output or internal);

Place - creates a place. Use selection mode to change the number of tokens;

1.3 Additional editor modes for circuit models

Input/output port - create a new input (Shift + left-click) or output (left-click) port;

Joint - create a new joint. Allows to branch wires from one source to multiple destinations;

Function - create a new function component.

2 Exercise 1: Dining philosophers

Consider the problem of dining philosophers. The problem states that there are 4 philosophers sitting around a table. Each philosopher either thinks or eats. There are four forks on the table shared among philosophers in such a way that each fork can be either used by a philosopher on its left or on its right side (see Figure 2).

As the problem states, before eating a philosopher tries to take both forks in some order. If a fork is occupied (by the neighboring philosopher) he will be waiting until the fork becomes available again, and then takes it. After eating, a philosopher returns to thinking while putting back both of his or her forks.

Petri Net model is ideal for depicting this behaviour. The corresponding Petri Net of this system is shown in Figure 3.

2.1 Task 1

Construct this model for three philosophers and try simulating it within Workcraft environment. It is better if you choose using STG model and dummy transitions for this purpose (select: File→Create Work...→Signal Transition Graph).

Try simulating this model by clicking on the button. You will see all of the enabled transitions will be marked. Try simulating the model and see the trace forming in the "Tool Controls" window.

2.2 Task 2

In the menu choose Tools→Verification→Check for deadlocks. This will execute the external program and produce a deadlock trace. Examine it using options available in "Tool controls" window.

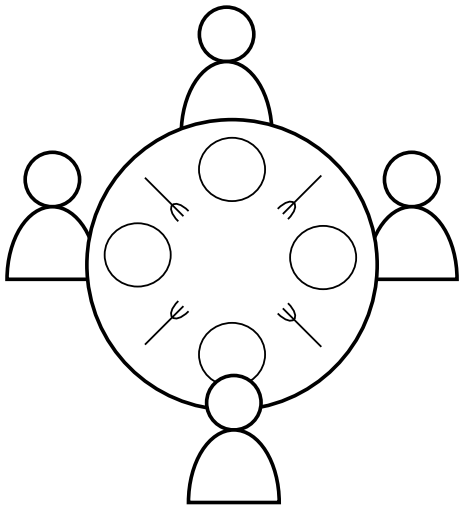


Figure 2: Dining philosophers

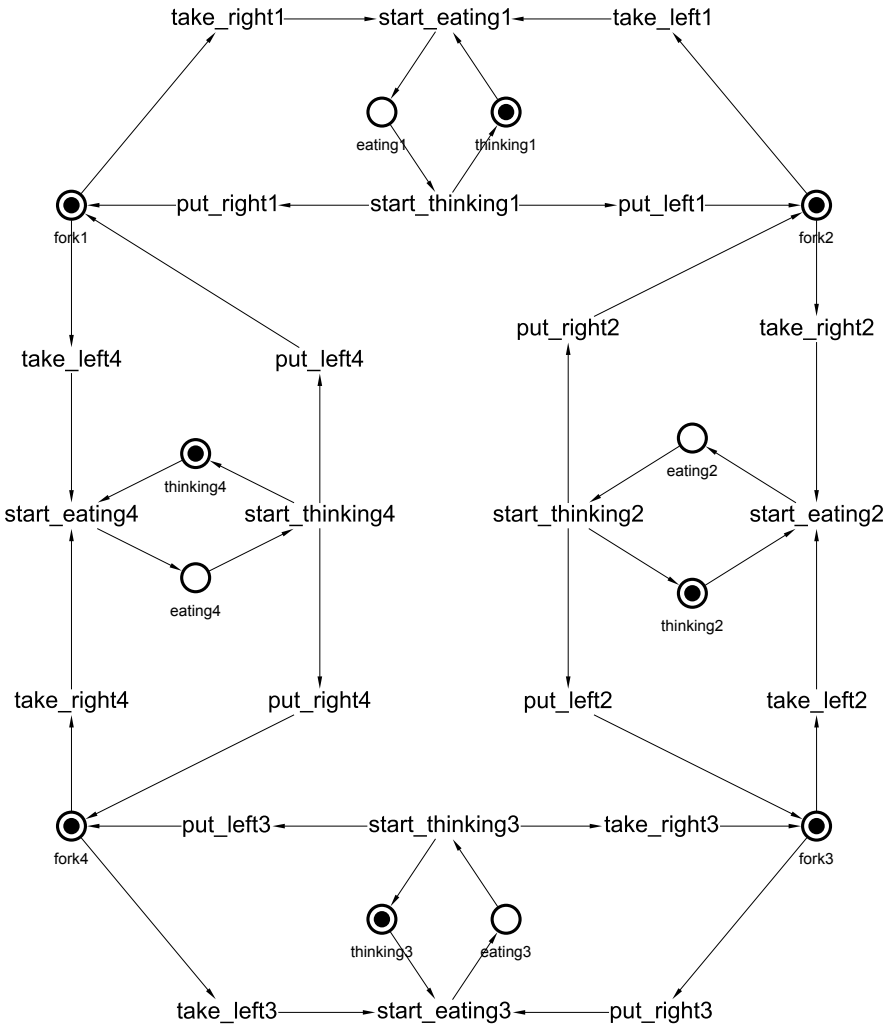


Figure 3: Dining philosophers (Petri Net)

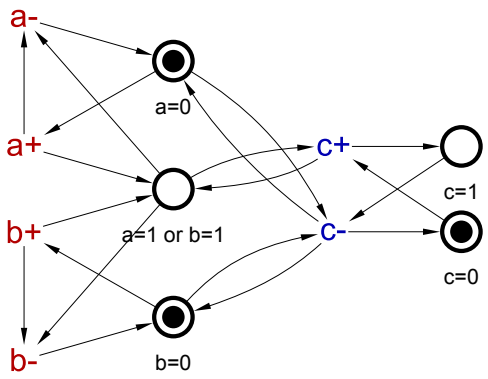


Figure 4: OR gate STG

2.3 Task 3

Modify the existing model in such a way that the deadlock never occurs (**hint:** simply make sure there are two philosophers in the system that take forks in particular order. One philosopher always takes his right fork first, while other starts with his left fork).

3 Exercise 2: Developing OR gate

Consider the example of an OR gate STG presented in Figure 4. It is a Petri net with transitions denoting changes in signal levels of a circuit which has two inputs a, b and one output c. The transition $c+$ is activated when either signal a or b is in active state. Notice how the *read arcs* are used to achieve the desired behaviour of c.

3.1 Task 1

Your first task it to model this STG and study its behaviour. You will notice that after both transitions $a+$ and $b+$ have fired, two tokens may appear in the place shared by $a+$ and $b+$ (all other places on this diagram will never have more than one token at a time). This is not the desired behaviour because most tools processing STGs only work with places not accumulating more than one token at a time (the 1-safe Petri nets).

3.2 Task 2

Your task is to modify the STG or create a new one describing the same behaviour of an OR gate and utilizing only places with not more than one token inside.

3.3 Task 3

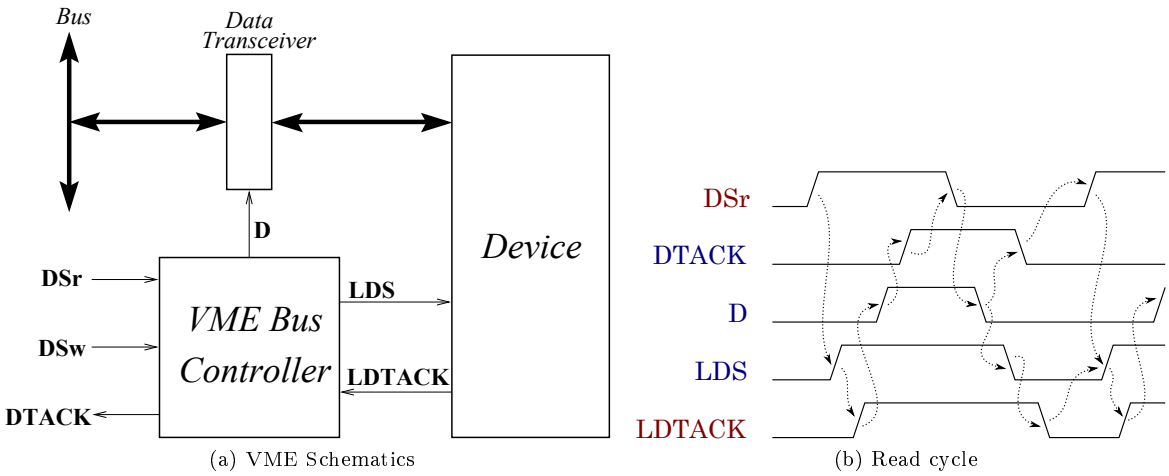
Try exporting the existing STG into an *or.g* file and synthesize it using *Petrify*. To export, use menu:

File→Export→.g (Workcraft STG serialiser).

Open the command line, enter the folder containing or.g. Enter the following command:

```
petrify or.g -nosi -eqn or.eqn
```

This will launch *Petrify* to synthesize the STG in terms of Boolean equations and save the result into file *or.eqn*. The *-nosi* option is needed in this example because the model is not speed-independent (signals a and b may change any time without allowing signal c to settle). If your STG was correct, *Petrify* will produce a file with a single equation: $c = a + b$.



(a) VME Schematics (b) Read cycle

Figure 5: VME bus controller

4 Exercise 3: VME bus controller

Figure 5a depicts the interface of a slave device to a VME bus. What is shown here is the result of an abstraction of the main synchronization core between the bus and the device links, separately from all remaining logic. The latter performs address and opcode decoding, error detection and some other functions that are outside this controller.

The behaviour of the controller is as follows: a request to read from or write into the device is received by one of the signals DSr or DSw respectively. In a read cycle, a request to read is sent to the device through signal LDS. When the device has the data ready (LDTACK), the controller must open the transceiver to transfer data to the bus (signal D, which is a Data Enable signal; it controls the transceiver together with a direction signal provided in the bus, namely it closes one latch and opens the tri-state in one direction, and opens the other latch in the other). In the write cycle, data is first transferred to the device by opening the transceiver (D). Next, a request to write is sent to the device (LDS). Once the device acknowledges the reception of the data (LDTACK) the transceiver must be closed to isolate the device from the bus. Each transaction must be completed by a return-to-zero of all interface signals, seeking for a maximum parallelism between the bus and the device operations.

4.1 Task 1

Construct signal transition graph for the VME read-cycle based on signal timing diagram in Figure 5b. The arrows in the diagram denote causal dependencies between the events.

4.2 Task 2

Export your model into *vme.g* file for further work with *Petrify*:

File→Export→.g (Workcraft STG serialiser).

Open the command line and enter folder containing the file you've exported. You can view the state space of the model by generating the state graph:

```
write_sg -bin vme.g > vme.sg
draw_astg -bin vme.sg > vme.ps
```

The *vme.ps* file will contain . Use *Petrify* to automatically solve CSC (complete state coding) conflict by introducing new signals (see the *-csc* option) and save the result into *vme2.g*. Call *write_sg* and *draw_astg* with *vme2.g* to produce *vme2.sg* and *vme2.ps*.

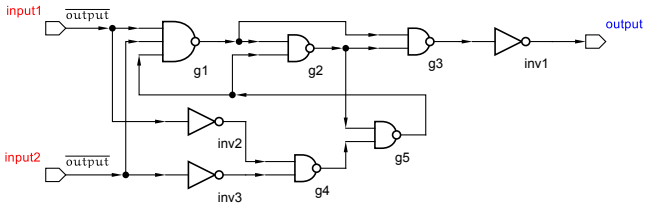


Figure 6: C-element decomposition

View *vme.ps* and *vme2.ps* to see how the conflict was resolved.

4.3 Task 3

Use *Petrify* to synthesize the *vme.g* file in complex gates (see the *-cg* option):

```
petrify vme.g -cg -eqn vme.eqn -no
```

Use *Petrify* to derive a netlist of gates by first decomposing the circuit into 3-input gates and then mapping onto a gate library (*-lit3 -tm*):

```
petrify vme.g -tm -lit3 -eqn vme2.eqn -no
```

Compare *vme.eqn* and *vme2.eqn*.

4.4 Task 4

Optimize the implementation (option *-topt*) by adding relative timing constraints to your STG model, you will need to use some text editor to change the *vme.g* contents directly.

There are several ways to include timing constrains in your *vme.g* file. All you need to do is make sure the signal transitions driven by the environment (red input transitions) are always slower than the transitions driven by the controller (blue output transitions). (**Hint**: study the *.slowenv* or *.slow* keywords in *Petrify* manual).

Obtain a complex gate implementation for the modified STG:

```
petrify vme.g -cg -eqn vme3.eqn -no
```

Compare it with the previous solution stored in *vme.eqn*.

5 Exercise 4: C-element decomposition

Consider the example of a C-element decomposition proposed by O. Maevsky. In this example the decomposition only consists of NAND gates and inverters (see Figure 6).

5.1 Task 1

In *Workcraft* environment start modelling a digital circuit, select: File→Create Work...→Digital Circuit. Proceed by creating several function components 

.

Each new function component will only have a single output contact, you will need to click on the contact to change its property. For defining simple combinational logic you only need to set the "Set Function" property to some Boolean expression. For instance, the expression $!(a + b)$ will specify the behaviour of a two-input NAND gate. As you input the expression, two new input contacts will be automatically created and the expression will appear next to the output contact (Figure 7).

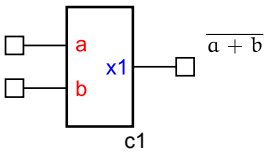


Figure 7: NAND gate example

After specifying contact behaviour, you can click on the component and set its "Render type" property to "Gate". This can improve visual presentation of your model.

Once components are created, create two input ports and one output port. The "Set Function" of the input signals will need to be set to the negation of the output: $\overline{output}$.

Finally, connect the components together and try simulating the circuit by clicking on the ▶ button. You will notice that all of the signals are inactive while some of the contacts are able to switch. Without switching the input port signals, continue clicking of the enabled contacts until all circuit signals have been settled. This simulation state will be the initial state of the model after clicking "Copy init" button in the "Tool controls" window.

5.2 Task 2

From the menu, select: Tools→Verification→Check for deadlocks and hazards.

This will launch circuit verification and check the circuit for deadlocks and hazards. A trace forming the hazard will be found. At the end of the trace there will be visible a signal transition which may get disabled by another transition. In the environment, where each of the gates may be arbitrary slow, this may looks like a glitch on one of the signals. Such glitches are hazardous for asynchronous circuits and must be avoided.

5.3 Task 3

Change the model so that the hazard disappears. You may use any two-input AND and OR gates as well as inverters.

5.4 Task 4

Decompose the three-input NAND gate into two-input gates, make sure new hazards are not introduced.