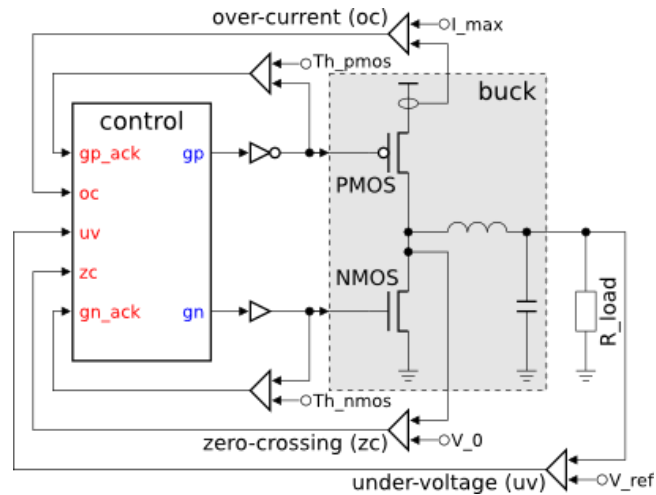


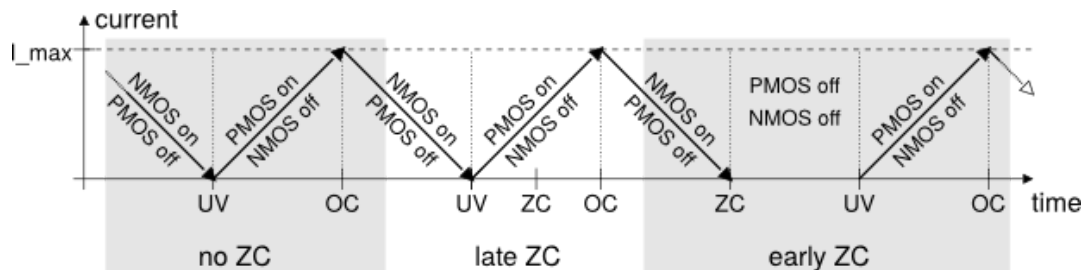
## Synthesis and verification of buck controller

Buck converter is a voltage step down and current step up converter. It comprises an analogue buck and its digital control logic as shown in the following diagram. Your task in this tutorial is to formally specify, synthesise and verify the control circuitry of the buck.



The controller switches the power regulating PMOS and NMOS transistors ON and OFF as a reaction to under-voltage (UV), over-current (OC) and zero-crossing (ZC) conditions. These conditions are detected and signalled by a set of specialised sensors implemented as comparators of measured current and voltage levels against some reference values ( $I_{max}$ ,  $V_{\theta}$ ,  $V_{ref}$ ). Note that the **gp** and **gn** signals are buffered to drive the very large power regulating transistors and their effect on the buck can be significantly delayed. Therefore, the controller is explicitly notified (by the **gp\_ack** and **gn\_ack** signals) when the power transistor threshold levels ( $Th_{pmos}$  and  $Th_{nmos}$ ) are crossed.

The operation of a buck is usually specified in an intuitive, but rather informal way, e.g. by enumerating the possible sequences of detected conditions and describing the intended reaction to them, as in the following phase diagram.



This specification reveals an alternation of the UV and OC conditions which are handled by switching the power regulating PMOS and NMOS transistors of the buck ON and OFF. Detection of the ZC condition after UV does not change this behaviour, however, if ZC is detected before UV then both the PMOS and NMOS transistors remain OFF until the UV event.

It is important to note that in order to avoid a short-circuit the PMOS and NMOS transistors of the buck must never be ON at the same time.

# Modelling

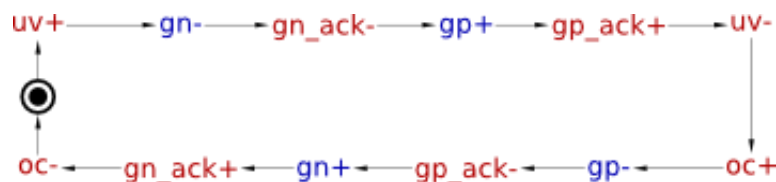
According to the phase diagram there are three distinctive scenarios to capture:

- **no ZC** – UV happens without ZC;
- **late ZC** – UV is followed by ZC;
- **early ZC** – UV happens after ZC.

Let us first capture the **no ZC** scenario as an STG.

- Initially the NMOS transistor is ON and the PMOS transistor is OFF which should lead to the UV condition:
  - Create a place  $p_0$  and mark it with a token - this denotes the initial state.
  - Create a rising phase of an input signal and call it  $uv+$ .
  - Connect the place  $p_0$  to the transition  $uv+$ .
- When UV is detected the NMOS transistor needs to be switched OFF:
  - Create an output transition  $gn-$ .
  - Connect  $uv+$  to  $gn-$ .
- Wait for indication of NMOS transistor being OFF:
  - Create an input transition  $gn\_ack-$ .
  - Connect  $gn-$  to  $gn\_ack-$ ,
- When the OFF state of NMOS is confirmed, PMOS transistor can be set ON to charge the buck:
  - Create an output transition  $gp+$  and an input transition  $gp\_ack+$ .
  - Connect  $gn\_ack-$  to  $gp+$  and  $gp+$  to  $gp\_ack+$ .
- Eventually the buck will saturate leading to reset of UV and OC conditions:
  - Create input transitions  $uv-$  and  $oc+$ .
  - Connect  $gp\_ack+$  to  $uv-$  and  $uv-$  to  $oc+$ .
- At this stage the PMOS transistor needs to be switched OFF:
  - Create an output transition  $gp-$  and an input transition  $gp\_ack-$ .
  - Connect  $oc+$  to  $gp-$  and  $gp-$  to  $gp\_ack-$ .
- After the OFF state of PMOS transistor is confirmed, NMOS transistor is switched ON state:
  - Create an output transition  $gn+$  and an input transition  $gn\_ack+$ .
  - Connect  $gp\_ack-$  to  $gn+$  and  $gn+$  to  $gn\_ack+$ .
- This leads to the release of OC and brings the controller to the initial state:
  - Create an input transition  $oc-$ .
  - Connect  $gn\_ack+$  to  $oc-$ .
  - Connect  $oc-$  to the place  $p_0$ .

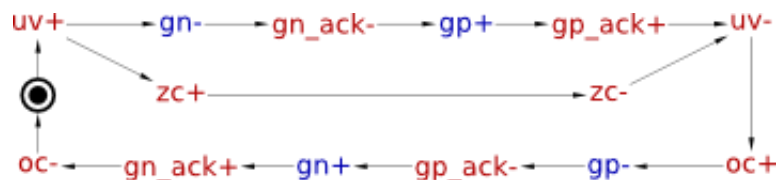
The resulting STG listing the sequence of signal events for this scenario is shown in the following diagram. Save this model as *stg-buck-scenario1\_no\_zc* file.



The scenario for *late ZC* is formalised in a very similar way. Both phases of ZC just happen concurrently with setting NMOS transistor OFF and PMOS transistor ON.

- Copy and save the *no ZC* scenario with new name *stg-buck-scenario2\_late\_zc*.
- Create two input signal transitions *zc+* and *zc-*.
- Connect *uv+* to *zc+*.
- Connect *zc+* to *zc-*.
- Connect *zc-* to *uv-*.

The resulting STG should look similar to the following diagram. Do not forget to save the work!



The scenario for early arrival of ZC is a bit different. Here the NMOS transistor needs to be switched OFF as soon as ZC is detected, without waiting for UV. However, switching the PMOS transistor ON is still delayed till UV condition.

- Copy and save the late ZC model under new name *stg-buck-scenario3\_early\_zc*.
- Delete incoming and outgoing arcs of *uv+* and *zc+* transitions (just select the arc and press Delete).
- Connect place *p0* to *zc+* and *zc+* to *gn-*.
- Connect *zc+* to *uv+* and *uv+* to *gp+*.
- Connect *gp+* to *zc-*.
- Rearrange transitions to make the STG look nicer (using the selection tool) and save the work.

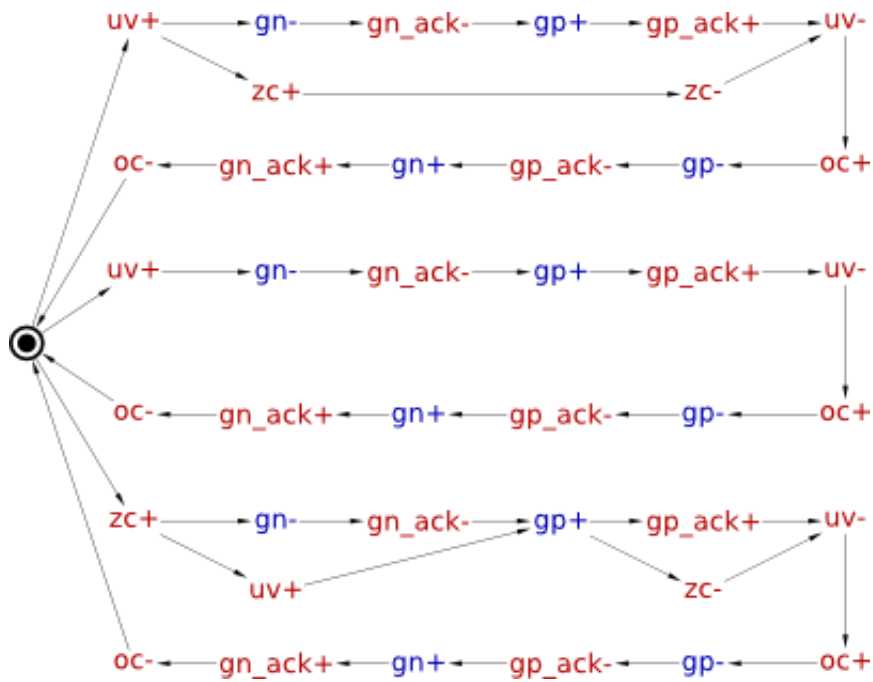
The STG for early ZC scenario should look similar to the following diagram.



In order to produce an implementation capable of handling all of the scenarios, these STGs need to be merged into a single specification. One can see that all three STGs have 'compatible' initial states, that is all common input and output signals are set to the same values initially. Therefore one can merge the initially marked place in the three STGs and obtain a combined specification for buck control.

- Create a new STG work called *stg-buck-scenarios\_merged*.
- Insert the STG for no ZC scenario by selecting *File*→*Merge work...* menu item and choosing the *stg-buck-scenario1\_no\_zc* file. After insertion the whole STG is selected - drag-and-drop it aside of the centre as the following steps will insert STGs there.
- Similarly insert the STG for early ZC scenario (*stg-buck-scenario3\_early\_zc* file) and drag it below the no ZC scenario.
- Finally insert the STG for late ZC scenario (*stg-buck-scenario2\_late\_zc* file) and drag it above the previously inserted ones.
- Now as you have STGs for all three scenarios in the same work space remove the initial place in two of the scenarios (e.g. in late ZC and early ZC) and reuse the remaining place instead.

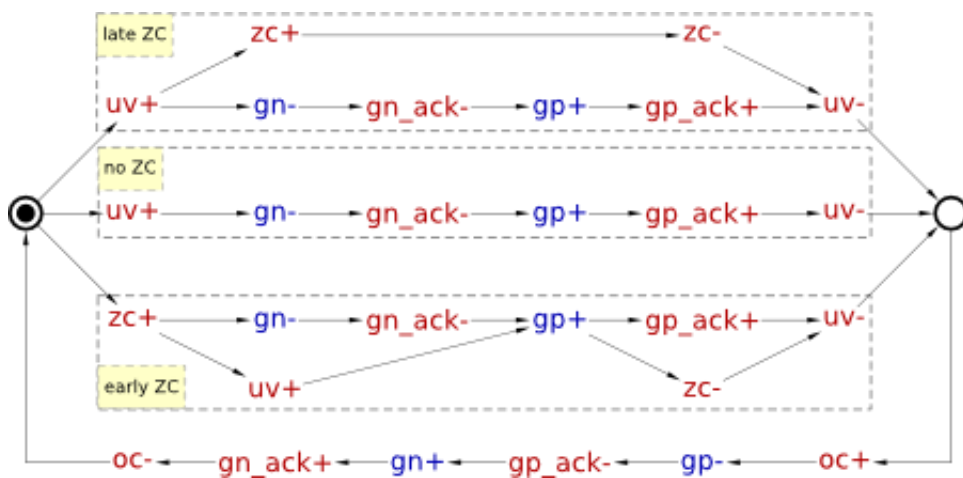
The STG combining all three scenarios should look like the following diagram. Save this STG.



Note that this STG is *non-deterministic*, e.g. after *uv+* fires the STG can end up in either of the two possible states. Non-determinism is common in scenario-based modelling. Occasionally, the scenarios may impose conflicting requirements on the system (e.g. two scenarios may have a common prefix, with one of them requiring and the other forbidding the circuit to produce a particular output after this prefix). Such conflicting requirements can be detected during verification. In this model, however, the scenarios are compatible.


### Optional simplification

Once the initially marked places are merged, one can notice that three transitions *oc-* leading to it can also be merged because their preceding states are ‘compatible’. This process continues with signal event *gn\_ack+*, and so on, ‘zipping’ the common paths of the STGs together. The simplified STG specification of the buck control is as follows; save it as *stg-buck-simplified* file.



Note that this STG is just a cosmetic improvement over the previous one and this step can safely be skipped. This does not affect the verification and synthesis, but it does improve the visual representation and thus is important from the designer's perspective.

## Verification of specification

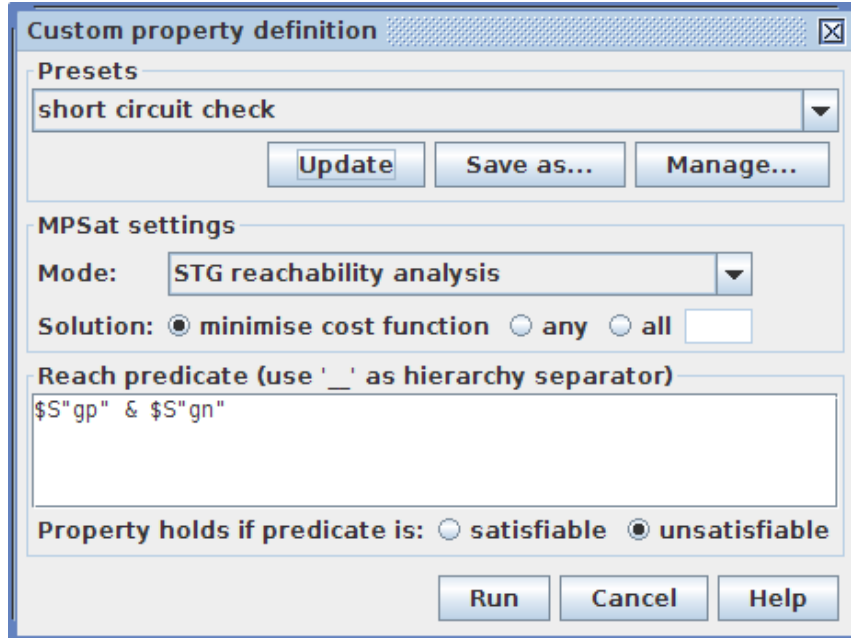
Activate the **simulation tool**  and exercise the obtained STG model. Click one of the enabled signal transitions (they are highlighted in orange) to *evaluate* the STG into the next state. Make sure the simulation traces correspond to those intended by the informal specification of the phase diagram.

Before proceeding to the synthesis step verify the specification for consistency (i.e. that the rising and falling phases of each signal alternate in all possible execution traces), deadlock-freeness and output-persistency. These can be done via *Tools*→*Verification* menu.

Another property one has to verify is that PMOS and NMOS transistors are never ON simultaneously (which would lead to a short-circuit), i.e. that signals **gp** and **gn** are never high at the same time. This custom property can be formulated as a reachability analysis problem using Reach language:

- Open the *Custom property definition* window by selecting *Tools*→*Verification*→*Custom properties [MPSat]...* menu.
- In *MPSat settings* set the *Mode* into *STG reachability analysis* and the *Solution* into *minimise cost function*.
- Enter a Reach expression that identifies the short-circuit, i.e. both gp and gn signals are high – `$$"gp" & $$"gn". 1)`.
- Select *unsatisfiable* to denote that the property holds if predicate is unsatisfiable.
- Save this property as a preset for future use, e.g. under the name *short circuit check*.

The whole custom property window should look as follows.



When you click the *Run* button the STG will be searched for a state where the Reach expression evaluates to *True*. If such a state exists then the Reach predicate is satisfiable and the property is violated. Otherwise, the property holds.

If the verified property is violated then a trace leading to the problematic state is reported. This trace can be simulated to diagnose the problem and correct it at the level of STG specification.

# Synthesis

The STG specification can now be synthesised into an asynchronous circuit implementation either with Petrify or MPSat backend tools via *Tools*→*Synthesis* menu.

A complex-gate solution obtained with Petrify (via *Complex gate [Petrify]* menu) is as follows (note that solution is not unique and you may get a slightly different set of equations):

```
[gp] = uv gn_ack' + gp_ack oc';  
[gn] = zc' uv' gp_ack';
```

Using De Morgan's law one can derive the following negative gate implementation:

```
[gp] = ((uv' + gn_ack) (gp_ack' + oc))';  
[gn] = (zc + uv + gp_ack)';
```

These equations can be mapped to complex-gates with the following functions:  $Z = ((A' + B) * (C' + D))'$  for **gp** and  $Z = (A + B + C)'$  for **gn**. Let us call the former gate *OAI2I2I* and the later *NR3*.

Circuit designers use hardware description languages, such as Verilog [<http://en.wikipedia.org/wiki/Verilog>] or VHDL [<http://en.wikipedia.org/wiki/VHDL>], to precisely describe the circuit. For example, the association of the ports to the gates' pins can be described by the following Verilog module (if you are not familiar with Verilog you can safely skip this part as it is not required by the rest of the tutorial):

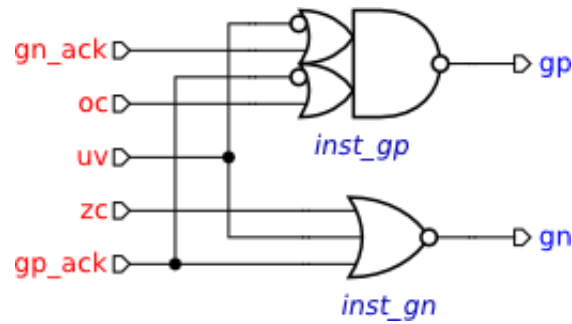
```
module control (oc, uv, zc, gp_ack, gn_ack, gp, gn);  
  input oc, uv, zc;  
  input gp_ack, gn_ack;  
  output gp, gn;  
  OAI2I2I inst_gp (.A(uv), .B(gn_ack), .C(gp_ack), .D(oc), .Z(gp));  
  NR3 inst_gn (.A(zc), .B(uv), .C(gp_ack), .Z(gn));  
endmodule
```

## Circuit capturing


Create a new Digital Circuit work called *circuit-buck-cg* and capture the implementation suggested by Petrify in the form of a gate-level netlist. In the future versions of Workcraft the derivation of a circuit from the synthesis output will be automated, but for now please do it manually.

- Create a Digital Circuit work *circuit-buck-cg*.
- Add a functional component with the set function  $((A' + B) * (C' + D))'$  <sup>2)</sup>. Rename it to `inst_gp` and change its rendering type to *GATE*.
- Add a functional component with the set function  $(A + B + C)'$  <sup>3)</sup>. Rename it to `inst_gn` and change its rendering type to *GATE*.
- Create two output ports **gp** and **gn**.
- Connect the output of `inst_gp` to the **gp** port and the output of `inst_gn` gate to the **gn** port.
- Create input ports **gn\_ack**, **oc**, **uv**, **zc** and **gp\_ack**.
- Connect the input port to the corresponding pins of the `inst_gp` and `inst_gn` gates. To fork a wire just start a connection from an existing wire - a joint point will be automatically created.
- Set the initial state of **gn** and **gn\_ack** signals to 1. For this select the corresponding ports and in the *Property editor* tick the *Init to one* check box.

The captured circuit should look as follows:



## Verification of implementation

Activate the **simulation tool**  and simulate the captured complex gate implementation of the buck control. Ports, pins and wires are colour-coded: blue means low level and red means high level of the signal. Excited pins and ports are highlighted in orange.

Click an excited pin to toggle its logical value – the circuit will evaluate to the next state where new set of signals will be enabled. The sequence of signal events is recorded in the simulation trace and can be subsequently replayed for analysing the circuit's behaviour.

Note that switching of input ports is not restricted. Environment can change them at any time causing unspecified behaviour of the circuit. One can restrict this behaviour by composing the circuit with the original STG specification of its contract with the environment:

- Activate **selection tool** and make sure nothing is selected – the Property editor will show the properties of the whole circuit.
- Click the *Environment URI* property - a file browser will pop up. Locate the *stg-buck-scenarios\_merged* file and open it. A path to that file will be copied to the *Environment URI* property <sup>4)</sup>.

Now circuit verification will be conducted in the context of the environment that behaves according to STG in *stg-buck-scenarios\_merged* file. Check the circuit for hazards, deadlocks and verify that it conforms to the environment specification. All these verification steps can be run via *Tools*→*Verification*→*Conformation, deadlock and hazard (reuse unfolding) [MPSat]* menu.

Try to alter the circuit and verify if it still conforms to the environment, is deadlock-free and operates without hazards.

## Solutions

Download all the Workcraft models discussed in this tutorial here:

[Buck control models \(26.02 KiB\)](#)

---

<sup>1)</sup> Here \$S means the value of a signal, "gp" and "gn" are the names of the signals and & is Boolean AND.

<sup>2)</sup>, <sup>3)</sup> Notice the use of ' symbol for negation.

<sup>4)</sup> If the file does not exist then its name is shown in red.