Using the BLESS proof tool to prove a simple thread behavior

Brian R Larson
brl@ksu.edu

February 7, 2017

Simple Example Using BLESS

This is a simple example of annotating a BLESS state machine so its proof outline can be transformed into an inductive proof.

Find the project “beginning_manage_regulator_status" to follow along using the BLESS proof tool.

The finished project, with tactic script, and proof is in project "proving_manage_regulator_status".

Manage Heat Regulator

The example thread was taken from a larger AADL project that modeled an infant incubator, or isolette.

The specification of behavior is taken from FAA's Requirement Engineering Management Handbook (REMH).

Specification from REMH

REQ-MRI-1: If the Regulator Mode is INIT, the Regulator Status shall be set to Init.

REQ-MRI-2: If the Regulator Mode is NORMAL, the Regulator Status shall be set to On.

REQ-MRI-3: If the Regulator Mode is FAILED, the Regulator Status shall be set to Failed.

Types.aadl

First, data types for ports must be created.

These are AADL data components, and could be located anywhere in the AADL project.

Other projects may be included by reference by right-clicking the AADL project folder (with an 'A' in the corner) opening "Preferences" and then selecting "Project References". Plugin_Resources is checked by default. Check other (open) projects to reference.

```
package Types
  public with Data_Model, BLESS;

data status --Regulator and Monitor Status
  properties
  Data_Model::Data_Representation => Enum;
  Data_Model::Enumerators => ("Init","O_n","Failed");
  BLESS::Typed => "enumeration (Init, O_n, Failed)";
end status;

data regulator_mode --Regulator Mode
  properties
  Data_Model::Data_Representation => Enum;
  Data_Model::Enumerators => ("INIT","NORMAL","FAILED");
  BLESS::Typed => "enumeration (INIT, NORMAL, FAILED)";
end regulator_mode;

end Types;
```

Both BLESS and Data Annex types are defined. `O_n` is used because "on" is an AADL keyword.

Thread Type

BLESS annex subclauses can be attached to AADL component types, but usually are put in implementations.

Our thread has one input `regulator_mode`, and one output `regulator_mode`, is periodic, dispatched every 100 ms.

```
thread manage_regulator_status
features
  regulator_status : out data port Types::status
    {BLESS::Assertion => "<<+=>REGULATOR_STATUS(x)>>";};
  regulator_mode : in data port Types::regulator_mode
      {BLESS::Assertion => "<<+=>REGULATOR_MODE(x)>>";};
properties
  Dispatch_Protocol => Periodic;
  Period => 100 ms;
end manage_regulator_status;
```

The ports have BLESS assertion properties to define their meaning. The +=> indicates the port has an enumeration type, so meaning is defined for each enumeration literal.

Assertion Annex Library

The BLESS assertions are defined in an Assertion annex library–unattached to any particular component.

```
annex Assertion
{**
<<REGULATOR_MODE:x+=>
  INIT   -> INI(),
  NORMAL -> REGULATOR_OK() and RUN(),
  FAILED -> not REGULATOR_OK() and RUN() >>

<<REGULATOR_STATUS:x+=>
  Init   -> REGULATOR_MODE(INIT),
  O_n     -> REGULATOR_MODE(NORMAL),
  Failed -> REGULATOR_MODE(FAILED)>>
**};
```

Assertions defining meaning for enumeration types also use the +=> symbol, and have enumeration literals with an arrow to what they mean. Assertions for `INI`, `REGULATOR_OK` and `RUN` are not needed for this example, but are used in the proof of the isolette system as a whole.

First Behavior without Assertions

```
thread implementation manage_regulator_status.impl
annex BLESS
  {**
    invariant <<true>>
    states
      start : initial state;
      run : complete state;
  transitions
    rs1: start -[ ]-> run {};
    rs2: run -[on dispatch]-> run
      {
      declare rm: Types::regulator_mode;
        {
        regulator_mode?(rm)
        ;
        if
        (rm=INIT)~> regulator_status!(Init)
        []
        (rm=NORMAL)~> regulator_status!(O_N)
        []
        (rm=FAILED)~> regulator_status!(Failed)
        fi
        }
      };
  **};
end manage_regulator_status.impl;
```

Don’t use Assertions on First Pass

Make a first state machine by intuition. An invariant is required; making it “true " doesn’t say anythine.

Two states: an `initial` state, and a `complete` state.

Two transitions: leave `initial` state, and do action when dispatched.

Need a local variable as target of port input from `regulator_mode`, followed by an alternative which choose what to output to `regulator_status`.

Invoke the Proof Engine

Click the praying hands icon to start the Proof Engine.

(You should enable the BLESS hot-key bindings, so you don't have to used the drop-down menu. Selections with hot-keys should show you the key combination for your platform.)

From the "BLESS" menu, choose "load model". This loads the whole model, and generates all the verification conditions, but keeps them in a buffer.

From the "BLESS" menu, choose "make an obligation" to pull the first verification condition (proof obligation) into the proof space.

First Proof Obligation

In the Eclipse Console, the first proof obligation should look like this:

```
[serial 1003]: Regulate::manage_regulator_status.impl
P [39] <<true>>
S [36]->
Q [36] <<true>>
  What for: <<M(run)>> -> <<I>> from invariant I when complete state run has Assertion <<M(run)>>
```

The assertion of each `complete` state must imply the `invariant`. In this case, state `run` has no assertion so gets the default, true.

First Tactic: axioms

Since true implies true (the arrow on the S means there is no action, so the proof obligation is P→Q), invoke the “axiom" group of tactics by BLESS -> Proof -> axioms.

```
This Proof Obligation:

[serial 1003]: Regulate::manage_regulator_status.impl
P [39] <<true>>
S [36]->
Q [36] <<true>>
Reason:  True Conclusion Schema (tc): P->true
  What for: <<M(run)>> -> <<I>> from invariant I when complete state run has Assertion <<M(run)>>

Has been solved by True Conclusion Schema (tc): P->true
```

The proof engine finds that True Conclusion Schema applies, removes it from the proof space, and fetches the next proof obligation to be solved.

Second Proof Obligation

```
[serial 1004]: Regulate::manage_regulator_status.impl
P [38] <<true>>
S [38]->
Q [38] <<true>>
  What for: Serban's Theorem:  disjunction of execute conditions leaving execution state start,
   <<M(start)>> -> <<e1 or e2 or . . . en>>
```

All execution states, and the initial state must always have an enabled outgoing transition (Serban's Theorem).

In this example, `start` has no assertion, and the transition leaving it has no transition condition (default true). It's proof obligation is easily solved by the same axiom tactic.

Transition `rs1`

Each transition gets a verification condition wherein the conjunction of the source state's assertion and the transition condition becomes the precondition of the transition's action, with the post condition being the destination's assertion.

```
    rs1: start -[ ]-> run {};
```

However, when the destination is a complete state, its assertion must hold on the next dispatch. So the destination's assertion is "aged" one period.

```
[serial 1005]: Regulate::manage_regulator_status.impl
P [38] <<true>>
S [41]->
Q [39] <<(true)^1>>
  What for:  <<M(start)>> -> <<M(run)>> for rs1:start-[ ]->run{};
```

Normalizing

BLESS -> Proof -> normalize invokes a host of tactics to put formulae into normal form. Applying normalize:

```
This Proof Obligation:

[serial 1005]: Regulate::manage_regulator_status.impl
P [38] <<true>>
S [41]->
Q [39] <<(true)^1>>
Reason:  Normalization
  What for:  <<M(start)>> -> <<M(run)>> for rs1:start-[ ]->run{};

  Normalization Axiom:

    Constants are always the same
Has been normalized to get:

[serial 1008]: Regulate::manage_regulator_status.impl
P [38] <<true>>
S [41]->
Q [39] <<true>>
  What for:   normalization of [serial 1005]
```

recognizing that `true` is a constant, so removes the `^1` and parentheses. Invoking the axioms tactic solves it.

Transition `rs2`

Transition `rs2` has an action, actually a composite action, which is reduced with BLESS -> Proof -> reduce composite.

```
[serial 1006]: Regulate::manage_regulator_status.impl
P [39] <<true>>
S [44]declare
  rm: Types::regulator_mode;
{
regulator_mode?(rm)
;
if
(rm = INIT)~>
  regulator_status!(Init)
[]
(rm = NORMAL)~>
  regulator_status!(O_N)
[]
(rm = FAILED)~>
  regulator_status!(Failed)
fi
}
Q [39] <<(true)^1>>
  What for:  <<M(run) and x>> A <<M(run)>> for rs2:run-[x]->run{A};
```

It's the Same!

Almost. Actually it reduced an asserted_action, to a behavior_action_block, but the change is not visible.

```
[serial 1010]: Regulate::manage_regulator_status.impl
P [39] <<true>>
S [44]declare
  rm: Types::regulator_mode;
{
regulator_mode?(rm)
;
if
(rm = INIT)~>
  regulator_status!(Init)
[]
(rm = NORMAL)~>
  regulator_status!(O_N)
[]
(rm = FAILED)~>
  regulator_status!(Failed)
fi
}
Q [39] <<(true)^1>>
  What for: as <<P>> S <<Q>> in <<P>> { S } <<Q>> [serial 1006]
```

So reduce composite again.

Proof Exception

Reduce composite caused a proof exception:

```
*****************************************
  PROOF  EXCEPTION
Mon Feb 06 16:16:52 CST 2017
*****************************************

from line 44:  "I'm trying to reduce existential lattice quantification to simpler
proof obligations and am trying to form <<P and x=e>> -> <<A>> from
<<P>> declare variable x:t:=e; { <<A>> T <<B>> } <<Q>>.
There is no <<A>>; put a true Assertion after the {"
*********************************
```

The proof engine views a behavior_action_block with a local variable as “existential lattice quantification" (see the LRM), and it can’t find something needed to reduce it. Fortunately, it tells you what it’s trying to do, and even makes a suggestion how to fix it.

The conjunction of the precondition, and the local variable having its initial value (except no initial value is used here), must imply the assertion just inside the block.

Add Needed Assertions

Put <<true>> just inside the beginning { and the ending }.

```
rs2: run -[on dispatch]-> run
  {
  declare rm: Types::regulator_mode;
    { <<true>>
    regulator_mode?(rm)
    ;
    if
    (rm=INIT)~> regulator_status!(Init)
    []
    (rm=NORMAL)~> regulator_status!(O_N)
    []
    (rm=FAILED)~> regulator_status!(Failed)
    fi
    <<true>> }
  };
```

Because the source was changed, you must reload, and then repeat the tactics. One you learn the hot keys, you can fire off the tactics in less than 5 seconds. The block reduces to three proof obligations.

Enter block, do block, exit block

```
[serial 1011]: Regulate::manage_regulator_status.impl
P [39] <<true>>
S [45]->
Q [45] <<true>>
  What for: <<P and x=e>> -> <<A>> in existential lattice quantification for [serial 1010]

[serial 1012]: Regulate::manage_regulator_status.impl
P [45] <<true>>
S [45]<<true>>
regulator_mode?(rm)
;
if
(rm = INIT)~>
  regulator_status!(Init)
[]
(rm = NORMAL)~>
  regulator_status!(O_N)
[]
(rm = FAILED)~>
  regulator_status!(Failed)
fi
<<true>>
Q [55] <<true>>
  What for: <<A>> T <<B>> in existential lattice quantification for [serial 1010]

[serial 1013]: Regulate::manage_regulator_status.impl
P [55] <<true>>
S [44]->
Q [39] <<(true)^1>>
  What for: <<B>> -> <<Q>> in existential lattice quantification for [serial 1010]
```

Solve the easy ones

Invoking “normalize" and “axioms" solves the block entry and exit proof obligations,

```
[serial 1016]: Regulate::manage_regulator_status.impl
P [45] <<true>>
S [45]<<true>>
regulator_mode?(rm)
;
if
(INIT = rm)~>
  regulator_status!(Init)
[]
(NORMAL = rm)~>
  regulator_status!(O_N)
[]
(FAILED = rm)~>
  regulator_status!(Failed)
fi
<<true>>
Q [55] <<true>>
  What for:   normalization of [serial 1012]
```

leaving a port input followed by an alternative.

Reduce Composite Again

makes the sequential composition (;) into two proof obligations.

There wasn't an assertion by the ; so the proof engine assumed `<<true>>`, but a ; will always need an assertion for proof.

```
[serial 1021]: Regulate::manage_regulator_status.impl
P [45] <<true>>
S [46]regulator_mode?(rm)
Q [45] <<true>>
  What for: <<P0>> S0 <<Q0 and P1>> in sequential composition for [serial 1016]

[serial 1022]: Regulate::manage_regulator_status.impl
P [45] <<true>>
S [48]if
(INIT = rm)~>
  regulator_status!(Init)
[]
(NORMAL = rm)~>
  regulator_status!(O_N)
[]
(FAILED = rm)~>
  regulator_status!(Failed)
fi
Q [55] <<true>>
  What for: <<Q0 and P1>> S1 <<Q1>> in sequential composition for [serial 1016]
```

Reduce Alternative

```
[serial 1021]: Regulate::manage_regulator_status.impl
P [45] <<true>>
S [46]regulator_mode?(rm)
Q [45] <<true>>
  What for: <<P0>> S0 <<Q0 and P1>> in sequential composition for [serial 1016]

[serial 1023]: Regulate::manage_regulator_status.impl
P [45] <<true>>
S [48]->
Q [48] <<(INIT = rm) or (NORMAL = rm) or (FAILED = rm)>>
  What for:   <<P>> -> <<B1 or B2 or ... or Bn>>  in if-[]-fi  for [serial 1022]

[serial 1024]: Regulate::manage_regulator_status.impl
P [48] <<true and (INIT = rm)>>
S [49]regulator_status!(Init)
Q [55] <<true>>
  What for:   <<P and B0>> S0 <<Q>>  for [serial 1022]

[serial 1025]: Regulate::manage_regulator_status.impl
P [48] <<true and (NORMAL = rm)>>
S [51]regulator_status!(O_N)
Q [55] <<true>>
  What for:   <<P and B1>> S1 <<Q>>  for [serial 1022]

[serial 1026]: Regulate::manage_regulator_status.impl
P [48] <<true and (FAILED = rm)>>
S [53]regulator_status!(Failed)
Q [55] <<true>>
  What for:   <<P and B2>> S2 <<Q>>  for [serial 1022]
```

Push and Make An

Each tactic (really tactic group) is applied to every proof obligation in the proof space.

When there are too many proof obligations for your taste or comprehension, you can

BLESS -> Actions -> push obligations back

to put them back on the unsolved proof obligation stack, initially loaded with verification conditions.

BLESS -> make an obligation

moves one to the proof space.

Reduce Atomic

Port input is an atomic action. It get reduced by

BLESS -> Proof -> reduce atomic

```
[serial 1021]: Regulate::manage_regulator_status.impl
P [45] <<true>>
S [46]regulator_mode?(rm)
Q [45] <<true>>
  What for: <<P0>> S0 <<Q0 and P1>> in sequential composition for [serial 1016]
```

which produces

```
[serial 1027]: Regulate::manage_regulator_status.impl
P [47] <<(true) and REGULATOR_MODE(rm)>>
S [47]->
Q [46] <<true>>
  What for: applied port input of value <<pre and rm=M(regulator_mode)>> -> <<post>> [serial 1021]
```

and is solved by BLESS -> Proof -> axioms

Proving Alternative

Reducing alternative (if-fi) produces a proof obligation for each guarded command, and one that insists that some guard will be enabled.

The next proof obligation moved to the proof space asks whether the precondition of the alternative implies some true guard:

```
[serial 1023]: Regulate::manage_regulator_status.impl
P [45] <<true>>
S [48]->
Q [48] <<(INIT = rm) or (NORMAL = rm) or (FAILED = rm)>>
  What for:   <<P>> -> <<B1 or B2 or ... or Bn>>  in if-[]-fi  for [serial 1022]
```

This asks us to prove something `<<(INIT = rm) or (NORMAL = rm) or (FAILED = rm)>>` from nothing `<<true>>` which is impossible.

Getting Recorded Tactic Script

Whenever a tactic does something it is recorded in `script.txt`. Everything sent to the console is recorded in `dump.txt`.

Although you can look at these in a text editor, the files are only written when a buffer has enough text.

To force the write buffers to be flushed to the file:

BLESS -> Actions -> close dump file

This flushes and closes both files.

Where's `script.txt`?

Eclipse puts `script.txt` and `dump.txt` in a system-dependent folder.

For Mac OS, this is in a "hidden" folder under the OSATE application:

```
closing dump.txt
closing dump.txt file "/Applications/JuneEclipse.app/Contents/MacOS/dump.txt"
closing script.txt file "/Applications/JuneEclipse.app/Contents/MacOS/script.txt"
Dump File Closed
```

Fortunately, it tells you where it is. You can continue to work on your proof, but henceforth the two files will not be updated–so it's often better to quit OSATE and launch it again.

Editing Your Script

On Mac OS, I like TextWrangler. It has an option in the Open dialog box to “show hidden items" so you can see the hidden folders. Generally, the part you want is from the last “load" to the end of file. I saved this in a new “scripts" folder in my project named: manage_regulator_script.script

```
load
#[serial 1003] <<M(run)>> -> <<I>> from invariant I when complete state run has Assertion <<M(run)
#Regulate::manage_regulator_status.impl
#[serial 1004] Serban's Theorem:  disjunction of execute conditions leaving execution state start,
make-an
axioms
#[serial 1005]  <<M(start)>> -> <<M(run)>> for rs1:start-[ ]->run{};
normalize
axioms
#[serial 1006]  <<M(run) and x>> A <<M(run)>> for rs2:run-[x]->run{A};
reduce
reduce
normalize
axioms
reduce
reduce
push
#Regulate::manage_regulator_status.impl
#[serial 1021] <<P0>> S0 <<Q0 and P1>> in sequential composition for [serial 1016
make-an
atomic
axioms
#[serial 1023]   <<P>> -> <<B1 or B2 or ... or Bn>>  in if-[]-fi  for [serial 1022]
```

More on Scripts

Comment line in script files start with #. When a proof obligation is moved to the proof space, its signature is put in a comment.

Also when “make-an" move one deliberately, it shows the name of its AADL component. This can be crucial for projects with many threads to know what’s being proved.

Only tactics that do something are recorded, so the “reduce atomic" that caused the exception is not recorded.

When you run scripts, three tactics that do nothing in a row will stop execution of the script. Click the praying hands icon to un-stop.

You can also insert “stop" which will stop script execution at that point, and then single-step through a troublesome part.

BLESS -> Actions -> show script terms will write a list of the script terms, and a brief description of what each does.

Running a script

BLESS -> get new script will open a file-choose dialog. Choose the script file you edited.

BLESS -> run script will execute the script (until the end, stop, or three ineffective tactics in a row).

Introducing an Axiom

Of course, `rm` is an enumeration type, `Types::regulator_mode`, so must be either `INIT`, `NORMAL`, or `FAILED`.

However, you can define a special kind of assertion, starting with `AXIOM`, that are assumed to be true. Be careful, because if you declare something false to be an axiom, your proof will be invalid. All user-defined axioms, and their use, must be specially scrutinized.

Using an Axiom

Put the definition of the axiom with the declaration of `rm`, and it use by the ; before the alternative.

```
declare rm: Types::regulator_mode
  <<AXIOM_RM: :(INIT = rm) or (NORMAL = rm) or (FAILED = rm)>>;
  { <<true>>
  regulator_mode?(rm)
  ;  <<AXIOM_RM()>>
  if
  (rm=INIT)~> regulator_status!(Init)
  []
  (rm=NORMAL)~> regulator_status!(O_N)
  []
  (rm=FAILED)~> regulator_status!(Failed)
  fi
```

Save the changes.

Remove Axioms from Postconditions

BLESS -> get new script

to select your script, then BLESS -> step script

until you get to

```
[serial 1021]: Regulate::manage_regulator_status.impl
P [46] <<true>>
S [47]regulator_mode?(rm)
Q [48] <<AXIOM_RM()>>
  What for: <<P0>> S0 <<Q0 and P1>> in sequential composition for [serial 1016]
```

BLESS -> Remove -> remove axioms from postconditions

so you won't need to prove the axiom holds after the port input.

Axiom Removed

```
[serial 1027]: Regulate::manage_regulator_status.impl
P [46] <<true>>
S [47]regulator_mode?(rm)
Q [48] <<true>>
  What for:  add user-defined axioms to postcondition:
  <<P>> S <<Q>>
 ---------------
  <<P>> S <<Q and A>>
```

The “What for" says you’re adding an axiom because the eventual proof will flow in the opposite direction.

Reduce atomic, and axioms as before to get:

```
[serial 1023]: Regulate::manage_regulator_status.impl
P [48] <<AXIOM_RM()>>
S [49]->
Q [49] <<(INIT = rm) or (NORMAL = rm) or (FAILED = rm)>>
  What for:   <<P>> -> <<B1 or B2 or ... or Bn>>  in if-[]-fi  for [serial 1022]
```

Substitute Assertion Labels

BLESS -> Substitute -> substitute Assertion labels

replaces `<<AXIOM_RM()>>` with its body:

```
[serial 1029]: Regulate::manage_regulator_status.impl
P [48] <<((INIT = rm) or (NORMAL = rm) or (FAILED = rm))>>
S [49]->
Q [49] <<(INIT = rm) or (NORMAL = rm) or (FAILED = rm)>>
  What for: substituted Assertions' predicates for  labels  [serial 1023]
```

Normalize to get rid of parentheses, then axioms to solve.

Next

The next proof obligation moved to the proof space is:

```
[serial 1024]: Regulate::manage_regulator_status.impl
P [49] <<(AXIOM_RM()) and (INIT = rm)>>
S [50]regulator_status!(Init)
Q [56] <<true>>
  What for:   <<P and B0>> S0 <<Q>>  for [serial 1022]
```

BLESS -> Proof -> reduce atomic

to reduce the port output.

Reducing Port Output

Reducing port output produces two proof obligations:

the precondition implies the port's assertion, given that the value will be the enumeration literal "Init",

and that the conjunction of the precondition and the port's having the value output implies its postcondition

```
[serial 1033]: Regulate::manage_regulator_status.impl
P [49] <<(AXIOM_RM()) and (INIT = rm)>>
S [50]->
Q [13] <<REGULATOR_MODE(INIT)>>
  What for: applied port output of enumeration type  regulator_status!(Init) [serial 1024]

[serial 1034]: Regulate::manage_regulator_status.impl
P [50] <<((AXIOM_RM()) and (INIT = rm)) and (regulator_status = Init)^0>>
S [50]->
Q [56] <<true>>
  What for: applied port output <<pre and (regulator_status=Init)^0>> -> <<post>> [serial 1024]
```

Solve the second proof obligation with axioms.

Enumeration Type Substitution

Recall the assertion of the port

```
regulator_status : out data port Types::status
  {BLESS::Assertion => "<<+=>REGULATOR_STATUS(x)>>";};
```

and the definition of `REGULATOR_STATUS`

```
<<REGULATOR_STATUS:x+=>
  Init    -> REGULATOR_MODE(INIT),
  O_n      -> REGULATOR_MODE(NORMAL),
  Failed -> REGULATOR_MODE(FAILED)>>
```

for the port output

```
regulator_status!(Init)
```

The proof engine determines that if “Init" is output, then `REGULATOR_MODE(INIT)` must hold.

Missing Information

However, this proof obligation is missing information. We know that the target variable of the previous port input, `rm` is `INIT`, but we don't know what that means.

```
[serial 1033]: Regulate::manage_regulator_status.impl
P [49] <<(AXIOM_RM()) and (INIT = rm)>>
S [50]->
Q [13] <<REGULATOR_MODE(INIT)>>
  What for: applied port output of enumeration type  regulator_status!(Init) [serial 1024]
```

Recall that reducing port input, we got some information we didn't use.

```
[serial 1027]: Regulate::manage_regulator_status.impl
P [46] <<true>>
S [47]regulator_mode?(rm)
Q [48] <<true>>
```

reduced to

```
[serial 1028]: Regulate::manage_regulator_status.impl
P [47] <<(true) and REGULATOR_MODE(rm)>>
S [47]->
Q [48] <<true>>
  What for: applied port input of value <<pre and rm=M(regulator_mode)>> -> <<post>> [serial 1027]
```

which was solved by axioms. But this has just the information we need:

REGULATOR_MODE holds for whatever enumeration literal was received by the in port.

Add that term to the assertion of ;

Adding Knowledge About Input

After adding `and REGULATOR_MODE(rm)` to the assertion by the ;

```
rs2: run -[on dispatch]-> run
  {
  declare rm: Types::regulator_mode
    <<AXIOM_RM: :(INIT = rm) or (NORMAL = rm) or (FAILED = rm)>>;
    { <<true>>
    regulator_mode?(rm)
    ;  <<AXIOM_RM() and REGULATOR_MODE(rm)>>
    if
    (rm=INIT)~> regulator_status!(Init)
    []
    (rm=NORMAL)~> regulator_status!(O_N)
    []
    (rm=FAILED)~> regulator_status!(Failed)
    fi
    <<true>> }
  };
```

Let's re-run our script

New Proof Obligation for Port Input

The new proof obligation with the added information is now

```
[serial 1021]: Regulate::manage_regulator_status.impl
P [46] <<true>>
S [47]regulator_mode?(rm)
Q [48] <<AXIOM_RM() and REGULATOR_MODE(rm)>>
  What for: <<P0>> S0 <<Q0 and P1>> in sequential composition for [serial 1016]
```

which reduces to

```
[serial 1027]: Regulate::manage_regulator_status.impl
P [47] <<(true) and REGULATOR_MODE(rm)>>
S [47]->
Q [48] <<AXIOM_RM() and REGULATOR_MODE(rm)>>
  What for: applied port input of value <<pre and rm=M(regulator_mode)>> -> <<post>> [serial 1021]
```

after removing axioms from postconditions, and normalizing

```
[serial 1030]: Regulate::manage_regulator_status.impl
P [47] <<REGULATOR_MODE(rm) and true>>
S [47]->
Q [48] <<REGULATOR_MODE(rm) and true>>
  What for:   normalization of [serial 1028]
```

which is solved by axioms

Port Output with Added Information

```
[serial 1024]: Regulate::manage_regulator_status.impl
P [49] <<(AXIOM_RM() and REGULATOR_MODE(rm)) and (INIT = rm)>>
S [50]regulator_status!(Init)
Q [56] <<true>>
  What for:   <<P and B0>> S0 <<Q>>  for [serial 1022]
```

which reduces to

```
[serial 1036]: Regulate::manage_regulator_status.impl
P [49] <<(AXIOM_RM() and REGULATOR_MODE(rm)) and (INIT = rm)>>
S [50]->
Q [13] <<REGULATOR_MODE(INIT)>>
  What for: applied port output of enumeration type  regulator_status!(Init) [serial 1024]

[serial 1037]: Regulate::manage_regulator_status.impl
P [50] <<((AXIOM_RM() and REGULATOR_MODE(rm)) and (INIT = rm)) and (regulator_status = Init)^0>>
S [50]->
Q [56] <<true>>
  What for: applied port output <<pre and (regulator_status=Init)^0>> -> <<post>> [serial 1024]
```

Axioms Laws Normalize

Applying axioms solves the second proof obligation.

BLESS -> Proof -> laws removes some unnecessary parentheses

BLESS -> Proof -> normalize removes more parentheses, and re-orders terms

```
[serial 1040]: Regulate::manage_regulator_status.impl
P [49] <<INIT = rm and AXIOM_RM() and REGULATOR_MODE(rm)>>
S [50]->
Q [13] <<REGULATOR_MODE(INIT)>>
  What for:   normalization of [serial 1038]
```

Adding Equivalent Terms

We know that `INIT = rm` and that `REGULATOR_MODE(rm)`, but we need `REGULATOR_MODE(INIT)`

BLESS -> Special -> add equivalent terms

```
[serial 1042]: Regulate::manage_regulator_status.impl
P [49] <<INIT = rm and AXIOM_RM() and REGULATOR_MODE(rm) and REGULATOR_MODE(INIT)>>
S [50]->
Q [13] <<REGULATOR_MODE(INIT)>>
  What for: Remove Equivalent Term: P(a) and P(b) and a=b is P(a) and a=b [serial 1040]
```

provides what we need. (The “What for" says “remove" because the completed proof will delete the equivalent term.)

Which is solved by the axiom: And Introduction Schema (aisph): (X and Y)->X

Same tactics solve other port output proof obligations

```
[serial 1025]: Regulate::manage_regulator_status.impl
P [49] <<(AXIOM_RM() and REGULATOR_MODE(rm)) and (NORMAL = rm)>>
S [52]regulator_status!(O_N)
Q [56] <<true>>
  What for:   <<P and B1>> S1 <<Q>>  for [serial 1022]
```

but reducing atomic raises a proof exception:

```
*****************************************
  PROOF  EXCEPTION
Tue Feb 07 08:21:57 CST 2017
*****************************************

from line 12:  "This label "O_N" in not among the choices in Assertion enumeration"
**********************************
```

Oops, we should have put out `O_n` instead. Fix the error, and re-run the script.

Use same tactic sequence on other port outputs

```
[serial 1025]: Regulate::manage_regulator_status.impl
P [49] <<(AXIOM_RM() and REGULATOR_MODE(rm)) and (NORMAL = rm)>>
S [52]regulator_status!(O_n)
Q [56] <<true>>
  What for:   <<P and B1>> S1 <<Q>>  for [serial 1022]
```

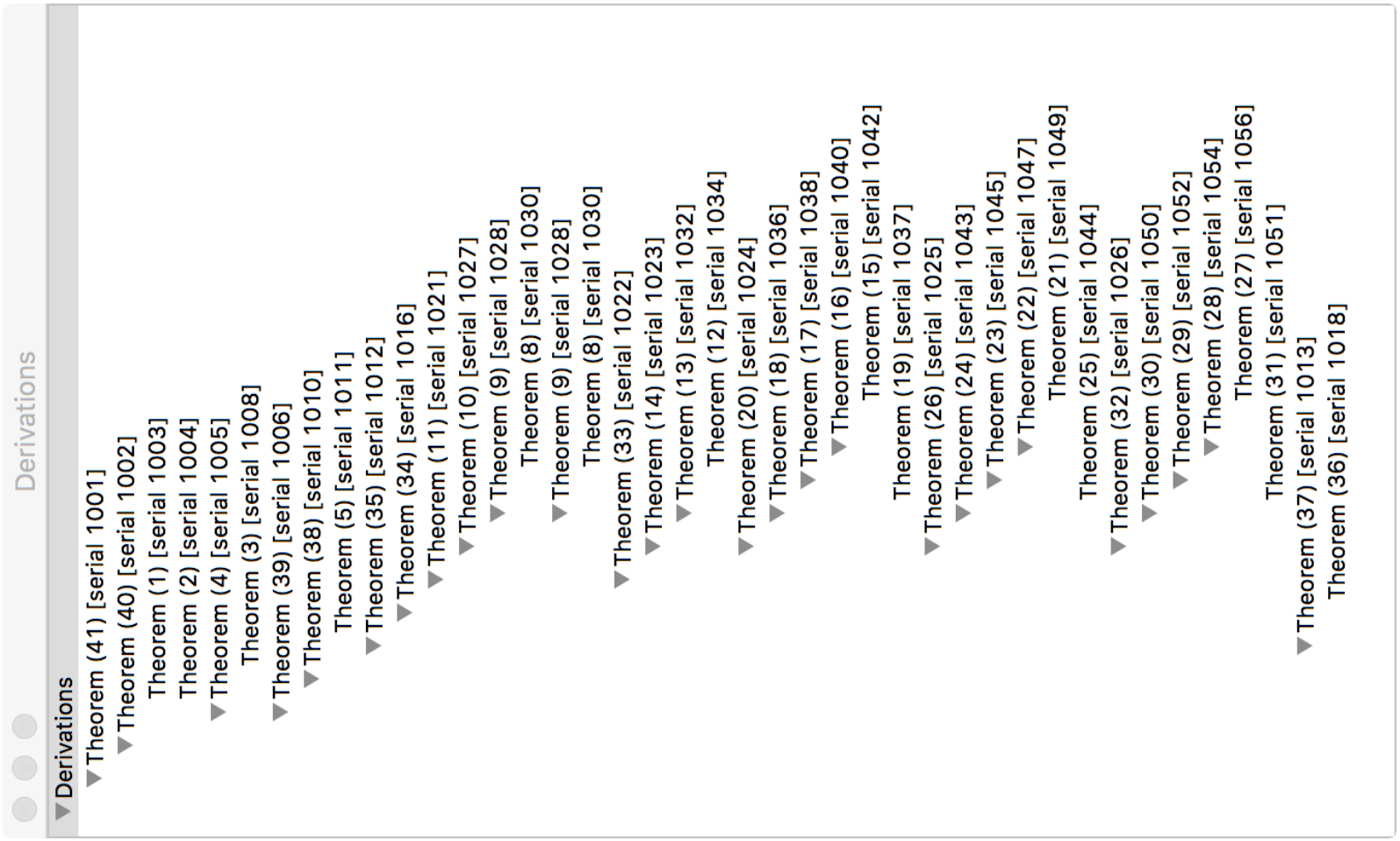
atomic axioms laws normalize equivalent axioms

solves the proof obligation above and the next

[serial 1026] for `regulator_status!(Failed)`

Proof!

When the last proof obligation is solved, the proof is created. Although formally the theorems are a sequence, effectively they form a tree:



40 Theorems

The last theorem, Theorem (41), only is meaningful when behaviors of multiple threads are proved.

Although ordered into theorems, the serial number remain the same. [serial 1003] was the first verification condition solved, that

```
 «M(run)» -> «I» from invariant I when complete
state run has Assertion «M(run)» in its
definition.
```

```
Theorem (1)                                [serial 1003]
P [39] <<true>>
S [36] ->
Q [36] <<true>>
by True Conclusion Schema (tc): P->true
```

The Last Theorem

Eventually, Theorem (40) uses Theorem (1)

```
Theorem (40)                                    [serial 1002]
P [36] << >>
S [36] ->
Q [36] <<manage_regulator_status.impl proof obligations>>
by Initial Thread Obligations
and theorems 1 2 4 39:
Theorem (1) [serial 1003] used for:
  <<M(run)>> -> <<I>> from invariant I when complete state run has Assertion <<M(run)>> in its def

Theorem (2) [serial 1004] used for:
  Serban's Theorem:  disjunction of execute conditions leaving execution state start, <<M(start)>>

Theorem (4) [serial 1005] used for:
   <<M(start)>> -> <<M(run)>> for rs1:start-[ ]->run{};

Theorem (39) [serial 1006] used for:
   <<M(run) and x>> A <<M(run)>> for rs2:run-[x]->run{A};
```

Theorems (1), (2), (4), and (39) come from the initial verification conditions determined during load.