EE382C: Verification and Validation
Project Final Report
Spring 2014

Name of Team Members: David Liu, Olamide Kolawole

Project Title: Modeling of Dual-Clutch and Semi-Automatic Transmission Logic in Alloy

Project Code Location: https://github.com/triskadecaepyon/DCT_Alloy_Model

Abstract:

Today's current transmission systems are much more complex than automatic transmissions of the past. With the removal of the torque converter in many currently produced automatic transmissions, the shift schedule, timing, and gear selection of a transmission are stricter; all the transmission parameters now have more influence to a car's drivability and resilience than ever before. For example, current transmission systems in the wrong gear may cause the engine to be out of its torque band, or may over-rev the engine, causing an engine failure.

Transmission engineers today must model more complex state machines than in the past, and take in inputs such as vehicle speed, engine speed, and gear set selection. Many system restrictions must be abided to, such as a shift schedule provided by other teams within the manufacturer. Information such as component stress constrains is provided to ensure certain stresses are never exceeded. Examples of stress constraints would be min/max clutch pressures, maximum torque allowed, and engine specifications on max RPM and torque curves.

Alloy's ability to model an abstracted state machine such as a transmission (before it gets coded and compiled) would be a great demonstration to the automotive industry of new practices to ensure great drivability and platform resilience. Standard industry tools typically default to Matlab and various extensions for modeling the transmission logic, but very few offer SAT checking capabilities, such as those found in a declarative language such as Alloy. This report looks to demonstrate the viability of using Alloy as a replacement modeler for current industry tools, and using the added SAT checker throughout development to shape the logic abstractions.

Introduction:

A transmission controller can by described fundamentally as a finite state machine. When automatic transmissions made the jump from analog to digital designs, more states and inputs were added, pushing a few manufacturers to figure out new methods of representing the states easily. Honda's patent, *System for controlling vehicle automatic transmission using fuzzy logic* (US 5323318 A), demonstrates a way of doing so using fuzzy logic. By giving inputs "fuzzy" definitions, Honda was able to port the simplified logic control to transmissions, allowing for easier design and testing, and faster time to market.

Honda’s simplified gear determination and state determination is demonstrated in the figures below, reproduced from their patent documentation.

U.S. Patent June 21, 1994 Sheet 11 of 16 5,323,318

FIG. 13

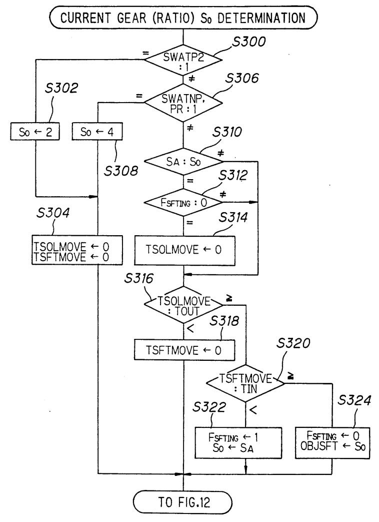
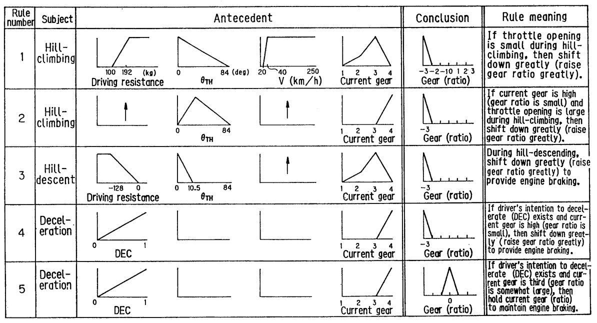


FIG. 5



Rule number	Subject	Antecedent				Conclusion	Rule meaning
1	Hill-climbing	100 192 (kg) Driving resistance	0 84 (deg) θ_{TH}	20 40 250 V (km/h)	1 2 3 4 Current gear	−3 −2 −1 0 1 2 3 Gear (ratio)	If throttle opening is small during hill-climbing, then shift down greatly (raise gear ratio greatly).
2	Hill-climbing		0 84 θ_{TH}		1 2 3 4 Current gear	−3 Gear (ratio)	If current gear is high (gear ratio is small) and throttle opening is large during hill-climbing, then shift down greatly (raise gear ratio greatly).
3	Hill-descent	−128 0 Driving resistance	0 10.5 84 θ_{TH}		1 2 3 4 Current gear	−3 Gear (ratio)	During hill-descending, shift down greatly (raise gear ratio greatly) to provide engine braking.
4	Decel-eration	0 1 DEC			1 2 3 4 Current gear	−3 Gear (ratio)	If driver's intention to decelerate (DEC) exists and current gear is high (gear ratio is small), then shift down greatly (raise gear ratio greatly) to provide engine braking.
5	Decel-eration	0 1 DEC			1 2 3 4 Current gear	0 Gear (ratio)	If driver's intention to decelerate (DEC) exists and current gear is third (gear ratio is somewhat large), then hold current gear (ratio) to maintain engine braking.

U.S. Patent June 21, 1994 Sheet 5 of 16 5,323,318

The current transmissions of today build upon that logic, but operate in a more complex manner. With the removal of the torque converter and planetary gear system (and their replacement with clutches), the clutch coupling is similar to that of a manual transmission car; the clutches are operated with hydraulic actuations of the gear shifting and clutch manipulation. This amplifies any logic or threshold issues, and can make the transmission operation undesirable.

The Alloy model:

Breaking ground on the Alloy model involved taking the basic elements of the transmission, turning them into object atoms, and threading together the basic relations that exist between them.

First iteration of the transmission code consisted of the following:

```
module base_transmission_model

sig Time {}

sig Gear {

       nextGear: lone Gear,

       previousGear: lone Gear

}


one sig Transmission {

       currentGear: Time -> lone Gear

}


fact connected {

       all g: Gear | Gear in g.^nextGear

       all g: Gear | Gear in
g.^previousGear

}


pred init (t: Time) {

}


pred noSameGear {

       all g: Gear | g.nextGear !in
g.previousGear

}


pred show {

       noSameGear

}


run show
```

The above code illustrates a few simple elements:

1. Individual atoms of gears
2. A single transmission casing (i.e. *one sig*)
3. All gears reachable in the set of gears from each other, either by *nextGear* or *previousGear* relations.

Running the Alloy execute command gives the following instance in Figure 1. One thing to note is that the loose fitting concept of next, previous, and current gears requires order, or individual instantiations of the relations to tie the gears together.

Running the system without the *fact connected* enabled, causes loose gear atoms to exist in the transmission.

The predicate, *noSameGear,* prevents the relation of nextGear and previousGear from pointing to the same atom.

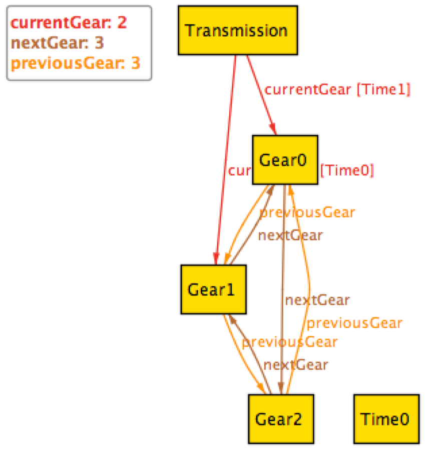


Figure 1

The first iteration of the model showed a few fundamental things:

1. The concept of order was difficult to represent in Alloy with the above method
2. The use of "time" atoms may not be beneficial for this Alloy model.

Some changes were made to the model to build upon the findings listed above.

```
sig Gear {

	nextGear: lone Gear,

	previousGear: lone Gear,

	shiftUp: one Shiftpoints,

	shiftDown: one Shiftpoints

}
```

The time atom was replaced with the relations of *ShiftUp* and *ShiftDown*, along with a new atom of a *shiftpoint*.

```
fact limitedGearsBySchedule {

	all t: Transmission | (Shiftpoints in t.gears.shiftUp) && (Shiftpoints in 
t.gears.shiftDown)

}

pred restrictedGearBox (t: Transmission) {

	//Says that the atoms of shiftschedules and shiftpoints reside in the 
Transmission

	// and cannot be "loose" atoms in the world.

	all s: ShiftSchedule | s in t.restricted

	all p: Shiftpoints | p in t.restricted.threshold

	all g: Gear | g in t.gears

}
```

New facts and predicates were used to prevent loose atoms from existing within the transmission, listed above. The full set of changes to render the file are available on the project code page. The updated model is shown in Figure 2, which reflects the changes listed above.

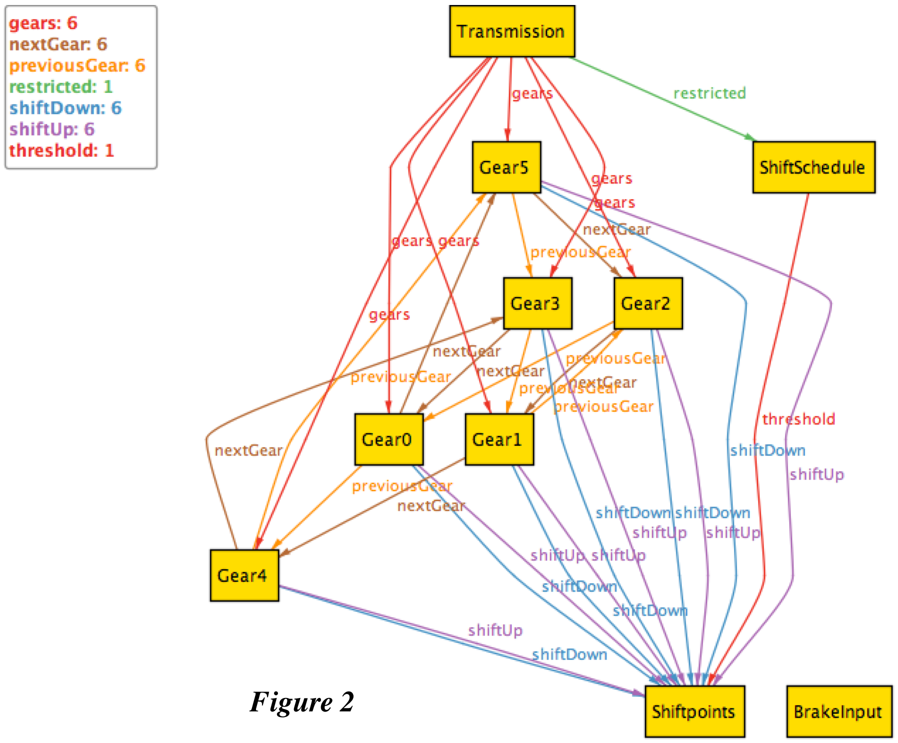


Figure 2

Notice the diagram in Figure 2, which is run with *run show for 1 but 6 gear* command. 6 gear atoms are instantiated, but the problem of instantiating the gears with the right relations to other gears (next/previous) would require a function signature - and again, the concept of ordering. What is present in this diagram, however, is the affixed gears within the transmission, shaping the rough relations of what the base transmission model should look like.

Alloy's Influence in shaping Logic Models

With the base Alloy model of the transmission created, it was time to move on and use Alloy's SAT checking abilities to help guide the rest of the modeling process.

It was found (through an answer by the professor) that the ordering method be explicit, and implemented by the *abstract sig* atoms. This means implemented gears of gear 1, gear 2, etc., that *extend* the gear interface. This modifies the initial code to look like the following:

```
abstract sig Gear {

       nextGear: lone Gear,

       previousGear: lone Gear,

       shiftUp: one Shiftpoints,

       shiftDown: one Shiftpoints

}

one sig GearR extends Gear {}

one sig Gear1 extends Gear {}

one sig Gear2 extends Gear {}
one sig Gear3 extends Gear {}

one sig Gear4 extends Gear {}

one sig Gear5 extends Gear {}
```

This allows for uniquely specified models transmission models, and allows for an exact match to a gear configuration in a transmission at test. In order to constrain the individual gears with the correct next/previous order, a predicate must be created and run:

```
pred gearRelations {

       GearR.nextGear = Gear1

       GearR.previousGear = GearR

}
```

```
Executing "Run show for 1"

   Solver=sat4j Bitwidth=4 MaxSeq=1 SkolemDepth=1 Symmetry=20

   1915 vars. 99 primary vars. 3258 clauses. 196ms.

   No instance found. Predicate may be inconsistent. 13ms.
```

However, running this predicate results in an issue. The reason that Alloy's instance checker returned no instances, stems from facts or included predicates being too strong. Analyzing the included facts and predicates, it was found that the *fact connected* expression was too strong.

```
fact connected {

       all g: Gear | Gear in g.^nextGear

       all g: Gear | Gear in g.^previousGear

}
```

The fact above states that *all* g of type Gear must be *reachable* from the nextGear or previousGear relations. With no previous predicate dictating next/previous relations, the move to abstract sig types for Gear broke the constraints of this fact. This fact had to be removed, as the gears were about to get explicit relations in the *gearRelations* predicate.

However, other predicates being added back showed other issues.

```
pred noSameGear {

       all g: Gear |g.nextGear !in g.previousGear

       //Says that no nextgear can be the same as its previous gear.

}
```

```
pred gearRelations {

       . . . . .

       Gear5.nextGear = Gear5

       Gear5.previousGear = Gear4

}
```

```
Executing "Run show for 1"

   Solver=sat4j Bitwidth=4 MaxSeq=1 SkolemDepth=1 Symmetry=20

   364 vars. 99 primary vars. 525 clauses. 39ms.

   No instance found. Predicate may be inconsistent. 2ms.
```

The above predicate also exhibits the same issue - no instance found. For the predicate of *noSameGear*, we again see that what worked to define the non-explicit Gear atoms do not hold up once explicit relations are enforced on next/previous gear relations.

In the predicate *noSameGear*, it stated that for any given Gear, no nextGear was the same as previousGear. This becomes a problem when you reach the end of a gear set's limits - there is no more gears past reverse, and once you reach top gear (in this case, 5th), no other gears can be achieved. With the explicit *gearRelations,* Gear5.nextGear = Gear5 breaks predicate *noSameGear (Figure 3)*.

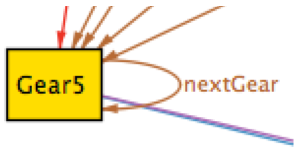


Figure 3

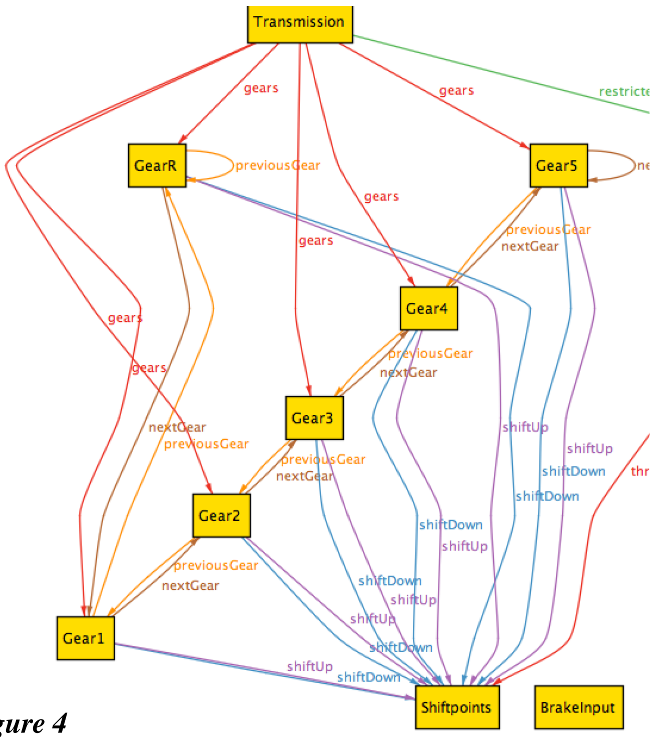


Figure 4

Now that the general model with ordering has been completed (Figure 4), it is now time to use Alloy to find other aspect of the transmission (such as shift points and shift schedules), and what relations are needed to make them work.

Leveraging Alloy's capabilities to define software abstractions and architecture

With the Gear relations in place, it is now time to look at the basic shift points, and their relations.

```
one sig ShiftSchedule {

        threshold: set Shiftpoints

        //A shiftschedule is made up of a
set of shift points

}



one sig S1 extends Shiftpoints {}

one sig S2 extends Shiftpoints {}

one sig S3 extends Shiftpoints {}

one sig S4 extends Shiftpoints {}

one sig S5 extends Shiftpoints {}
```

```
pred shiftRelations {

        GearR.shiftUp = S1

        Gear1.shiftUp = S2

        Gear2.shiftUp = S3

        Gear3.shiftUp = S4

        Gear4.shiftUp = S5

        Gear1.shiftDown = S1

        Gear2.shiftDown = S2

        Gear3.shiftDown = S3

        Gear4.shiftDown = S4

        Gear5.shiftDown = S5

}
```

The above shift points are a guess at the minimum required to use in the above 5 gear + 1 reverse Transmission model.

The biggest issue found is what Gear5's shift up and Gear Reverse's shift down had no relation tied to them. In this instance, Alloy is showing multiple instances that exist; some not pointing anywhere, other times pointing to a shift point of which it is not assigned to (Figure 5 and 6).

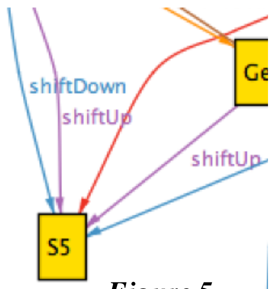


Figure 5

At this point, Alloy is using its instance checking abilities to show that the model has an insufficient amount of shift points to the amount of gears that it has.

Looking at this from an software abstraction standpoint, an object definition for the amount of shift points to be instantiated would be *Gears + 1 = number of shift points*. From a relations standpoint, an unreachable *shift point ceiling* and *shift point floor* must represent the first and last shift point, since the two "floating" limits need to match the abstracted Gear model (Figure 7).

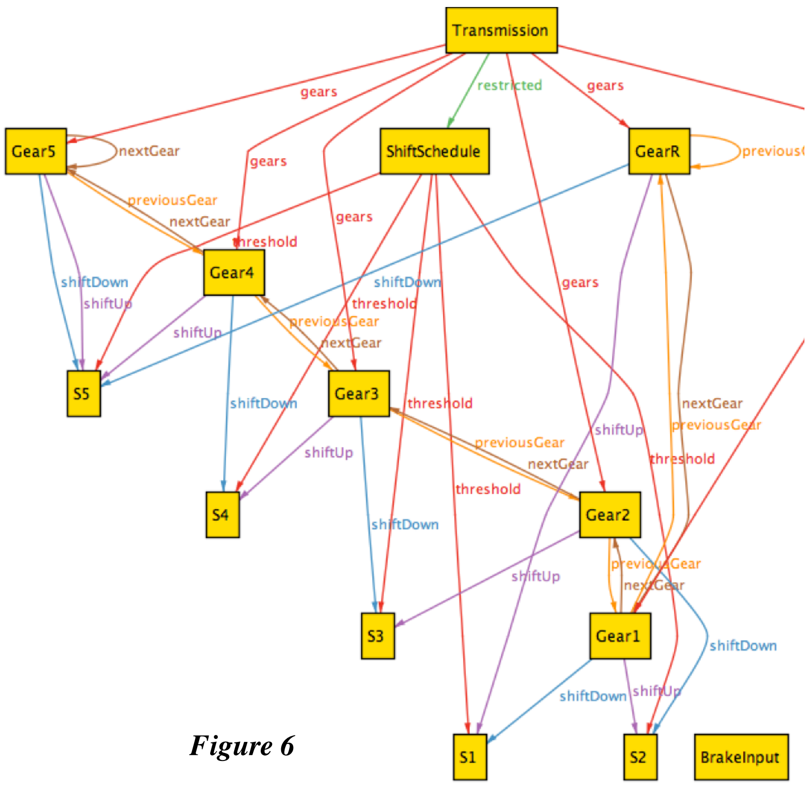


Figure 6

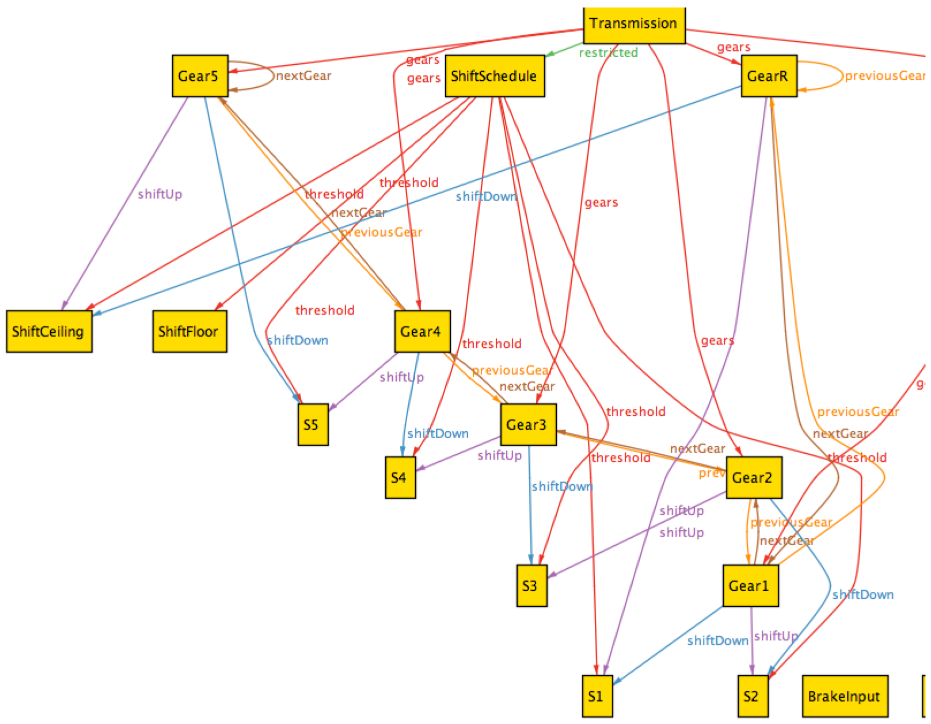


Figure 7

```
one sig S5 extends Shiftpoints {}

one sig ShiftFloor extends Shiftpoints {}

one sig ShiftCeiling extends Shiftpoints {}


pred shiftRelations {

       GearR.shiftUp = S1

       Gear1.shiftUp = S2

       Gear2.shiftUp = S3

       Gear3.shiftUp = S4

       Gear4.shiftUp = S5

       Gear5.shiftUp = ShiftCeiling

       GearR.shiftDown = ShiftFloor

       Gear1.shiftDown = S1

       Gear2.shiftDown = S2

       Gear3.shiftDown = S3

       Gear4.shiftDown = S4

       Gear5.shiftDown = S5

}
```

So now, we find that the updated model is correct, we have great software abstractions and architecture that we may port into our embedded system language of choice.

```
fun shiftUp [t: Transmission, g: Gear] : set Gear {

       //Needs Threshold interaction here.

       t.gears.nextGear

}


assert everyShiftUpDefined {

       all t: Transmission, g: Gear | g !in shiftUp[t,g] && show

}
```

```
pred show {

        all t: Transmission | noSameGear && restrictedGearBox[t]

//Display with the following restrictions.

}


check  everyShiftUpDefined
```

The above shows a template to check the function of shifting up in a transmission. Running the check results in a few instances being found, which means that our predicates and other relations are not extensive enough to create a sound model.

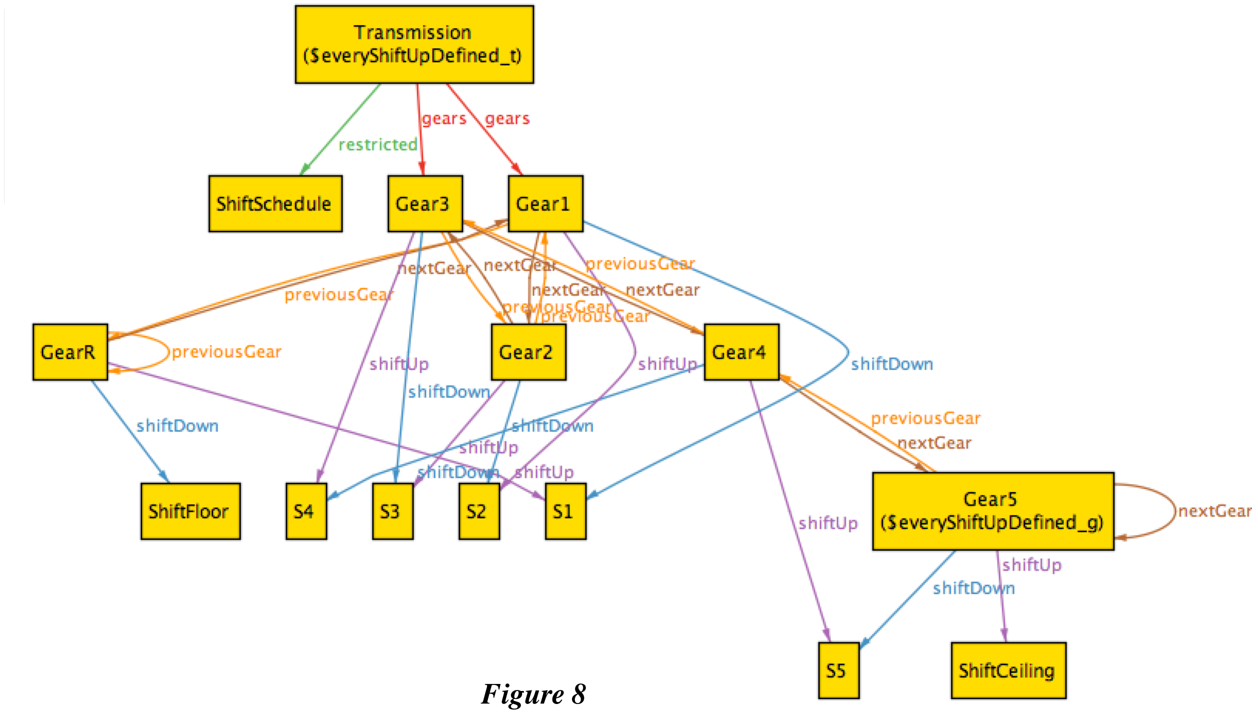


Figure 8

From the model above in Figure 8, we can see that there exists several of the gears that do *not* reside in the definition of the transmission. This can be problematic, as our function of *shiftUp* (both abstracted and software definition wise) relies on the collection of gears defined within it.

To fix this, we must first change the *fact connected* definition:

```
fact connected {

        all t: Transmission, g: Gear | g in t.gears

}
```

And then change the *everyShiftupDefined* assertion to match:

```
assert everyShiftUpDefined {

      all t: Transmission, g: Gear | g in shiftUp[t,g] && show || g in t.gears

}
```

The reason for the change in the assert is because GearR has no nextGear pointing to it. This could be solved by changing the *abstract sig* implementation of gear, or by saying that the gear must exist in the gear set within the transmission.

```
Running the analyzer again, gives:

Executing "Check everyShiftUpDefined"

   Solver=sat4j Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20

   862 vars. 211 primary vars. 1341 clauses. 79ms.

   No counterexample found. Assertion may be valid. 18ms.
```

Which demonstrates that are relations and constraints are relatively sound, at least in their current implementation.

Piping Alloy output for alternate display options

For many transmission engineers, the visual output of Alloy may not suit the needs of their documentation requirements. To better display the instances and counterexamples from Alloy, a third party display solution was adapted to Alloy's XML output.

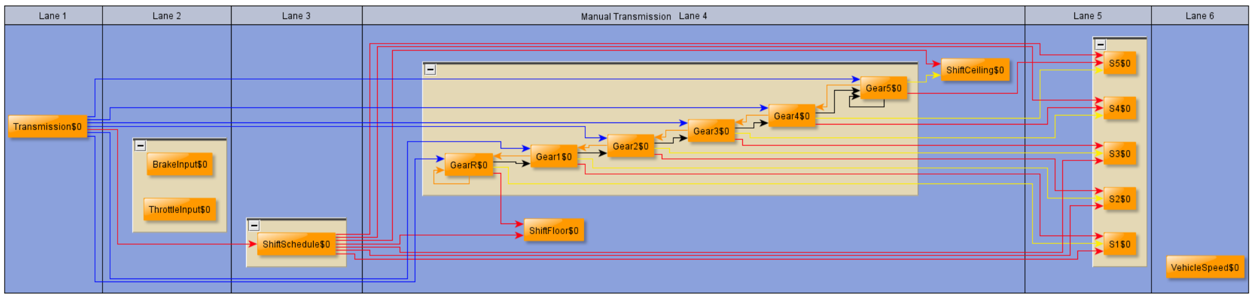


Figure 9

The solution, *YeD* (http://www.yworks.com/en/products_yed_about.html*)*, allows for custom organization, swim lanes, and groupings to be made, based upon a configuration file (Figure 9). An explanation of how it was adapted is documented below.

For the graphical display, the Alloy XML output had to be decomposed into a node-edge nomenclature. The alloy XML comprises of mainly two XML elements that are useful as input for graphical representation; *Sig* and *Field*.

The attributes of the *Sig* elements are described as follows:

Label = Name of the signature

ID= XML id to accurately identify a sig

parentID = ID of the parent Sig

The *Sig* encapsulates the number of instances of itself with atom elements. Each atom elements represents an instance of the Sig of an Alloy output.

The Field *node* contains relationships between the model's *Sig* elements. Similar to the *Sig* elements, the number of children represents the number of relationships. The attributes of the Field node is as follows.

Label = Name of the Field or relationship

ID= XML id to accurately identify a Field

parentID = id of the parent Sig.

Each instance of a relationship is encapsulated within a tuple node. In a Tuple node, it can hold any number of atoms to denote a relationship between said child nodes. Also there is a "types" element that is used to denote what *Sig* elements are participating in each relationship. For more complex analysis, the "types" elements could be used to code additional functionality depending on the variant of relation is embedded within field elements. Here is a sample *Field* and *Sig* node setup:

```
<sig label="base_transmission_model/Gear" ID="14" parentID="2">

   <atom label="base_transmission_model/Gear$0"/>

   <atom label="base_transmission_model/Gear$1"/>

   <atom label="base_transmission_model/Gear$2"/>

</sig>

<field label="nextGear" ID="15" parentID="14">

   <tuple> <atom label="base_transmission_model/Gear$0"/> <atom
label="base_transmission_model/Gear$2"/> </tuple>

   <tuple> <atom label="base_transmission_model/Gear$1"/> <atom
label="base_transmission_model/Gear$0"/> </tuple>

   <tuple> <atom label="base_transmission_model/Gear$2"/> <atom
label="base_transmission_model/Gear$1"/> </tuple>

   <types> <type ID="14"/> <type ID="14"/> </types>

</field>
```

The definition of the XML output had to be hypothesized as it was difficult to find documentation or specifications the Alloy XML output. Throughout the development process, varying output would break previous assumptions of the data representations in the XML. After a few repeated tries, a more standard definition for the transmission model's Alloy XML was created. The "$" symbol is not present in the actual visual output from Alloy, it is only present in XML exported instance file.

For input into the graphical display, editors with the capability of using GraphML as an input were used. GraphML is an XML-based file format used to depict graphs. A GraphML file contains structural properties that are used to describe instances of objects and the topology that exist within it. A basic GraphML file is outlined below:

```
  <graph id="G" edgedefault="undirected">

    <node id="n0"/>

    <node id="n1"/>

    <edge id="e1" source="n0" target="n1"/>

  </graph>
```

The GraphML above shows that a node called *n0* is connected to another node *n1*. For the purpose of this project, we mapped the *Sig* elements in Alloy XML to node elements in GraphML; tuple elements encapsulated by a field element in Alloy XML are mapped to edge elements in GraphML.

The yEd Graph editor uses GraphML as its native file input, and is able to re-arrange nodes on a graph with better organization. Although GraphML is its main input, it can also accept formats such as Microsoft Excel and other variants of XML files. yEd also provides libraries in platforms like Java, .Net and Android for more customized diagrams.

For the graphing, it was decided to transform the Alloy XML output into XML, with the help of a style sheet. The XSLT incorporates some of yEd's functions to transform the desired output from the XML. This allows the visual properties to be embedded in the GraphML before loading the file into a Graphing software. The transmission model presented in this report was used during testing; further modifications will have to be made to suit other models, since naming and organization changes are specific to each model. A snippet of the XSLT file is below.

```
<xsl:template match="//*" mode="create-edges">

             <xsl:element name="edge">

                    <xsl:attribute name="id">

        <xsl:value-of select="generate-id()" />

      </xsl:attribute>

                    <xsl:attribute name="source">

        <xsl:value-of select="./atom[1]/@label" />
      </xsl:attribute>

                    <xsl:attribute name="target">

        <xsl:value-of select="./atom[last()]/@label" />

      </xsl:attribute>

<data key="d1">

            <y:PolyLineEdge>

                    <xsl:choose>

                          <xsl:when test="../@label = $shiftUp">

                                <y:LineStyle type="line" width="1.0"
color="#ffee00" /> <!-- yellow -->

                          </xsl:when>

                          <xsl:when test="../@label = $gears">

                                <y:LineStyle type="line" width="1.0"
color="#0009ff" /> <!-- blue -->

                          </xsl:when>

                          <xsl:when test="../@label = $nextGear">

                                <y:LineStyle type="line" width="1.0" color="00ff88" /
> <!-- green -->

                          </xsl:when>

                          <xsl:when test="../@label = $powered">

                                <y:LineStyle type="line" width="1.0"
color="#ee00ff" /> <!-- magenta -->

                          </xsl:when>

                          <xsl:when test="../@label = $previousGear">

                                <y:LineStyle type="line" width="1.0"
color="#ff8d00" /> <!-- orange --

                          </xsl:when>

                          <xsl:otherwise><y:LineStyle type="line" width="1.0"
color="#FF0010" /></xsl:otherwise>

                    </xsl:choose>

                    <y:Arrows source="none" target="default" />

                    <xsl:element name="y:EdgeLabel">

                          <xsl:value-of select="../../@label" />
                    </xsl:element>

                </y:PolyLineEdge>

        </data>

</xsl:element>

</xsl:template>
```

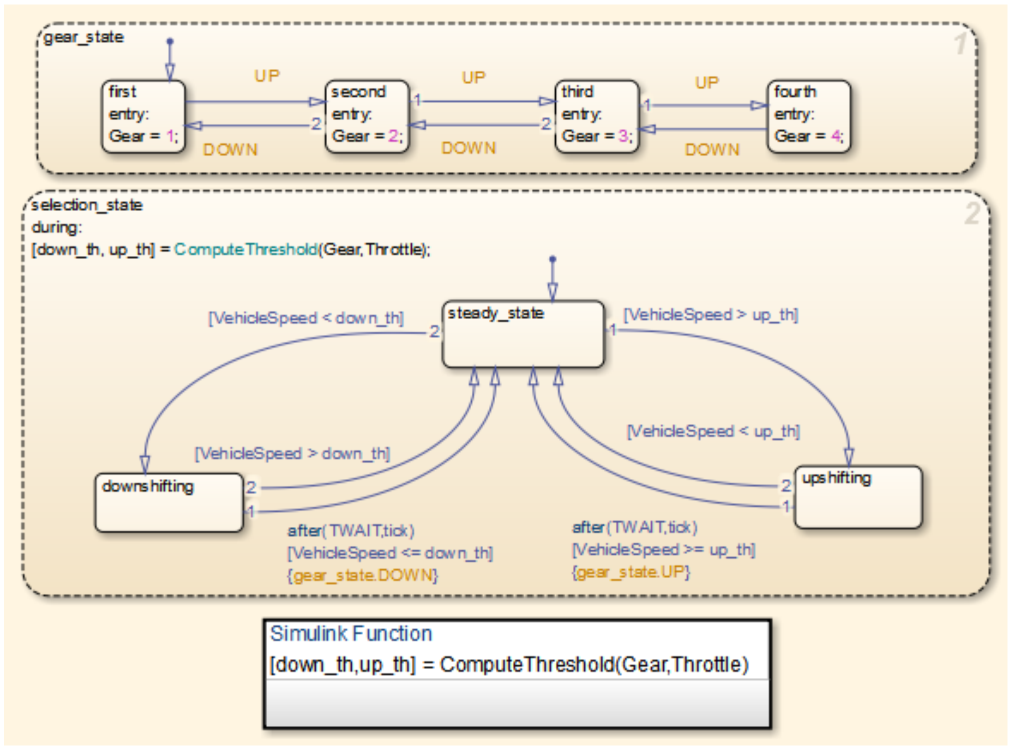
For the output design style, a swim lane design was chosen for easier analysis. The transmission model has sequential flow, accented with stages or levels of execution. With yEd's capabilities, the nodes are arranged automatically in the desired format.

With this, the output model from the yEd adjusted Alloy output is better understood than the stock Alloy output. Nodes of the same Alloy *Sig* are grouped together with uniform lines running across the graph, and each group is separated to different swim lanes to easily distinguish the transmission logic design.

Although the yEd was the chosen graph editor for this report, any other graph editor that can take GraphML as input could be used. This implementation example should allow the Alloy output to be an used easily as a model display for the transmission design, customizable to each manufacturer's needs. With the adapted, domain specific display, one could easily place Alloy and an adjusted graphing output into their workflow.

Use cases for Alloy in transmission logic design

The industry workflow of transmission logic design typically starts by referencing older, hand-drawn models, or by creating models in a modeling software, such as Matlab/Simulink.



The above example comes from the Matlab/Simulink example of *Modeling an Automatic Transmission Controller*, and demonstrates a simplified model to showcase its capacity to assist in model creation. Once a transmission model is created, it then is put through a test suite of known road simulations, and then subsequently ported to code. Once compiled, the code is then tested on a road dynamometer, and then tested on open roads.

The biggest issue with current, non-Alloy based workflows are:

1. Finding ways of checking for Model inconsistencies without heavily modifying the model code

2. Creating tests from the model instead of creating tests from past transmission data (i.e. white box testing)

3. Checking the Model for soundness if the configuration is changed (i.e. 7 gears instead of 6, dual clutches instead of single)

With the processes outlined in this report, one could use Alloy to thoroughly check the model, as well as create tests from the model with relative ease. By creating a general model first, different configurations of transmissions could be automatically tested and checked with assert statements, and then the subsequent final model's definition can be used as a template for the architecture (or automated architecture, for that matter).

The assert definitions could be restructured into an automatic tests for the final software model, which can prevent at least a few of the human induced errors that come with software porting effort.

With a bit more work, the Alloy language and assets could be used to replace Matlab/Simulink and other related tools in the automotive engineering world.

Final thoughts on the Alloy's viability in transmission logic design

Transmission design, along with many other finite state machine driven designs, have had nearly the same design processes for decades. Tools such as Matlab have been used in industry to assist in the state machine design, and many additional toolsets that build on Matlab (which are quite expensive) exist because there has not been much deviance from the current industry workflow.

Alloy's capabilities as a SAT checker and declarative language give it the potential to be used as either a replacement toolset, or additional toolset in the industry workflow. Based on the investigative work detailed in this report, it is believed that with a bit more effort, a general transmission model can be created, tested, and adopted by the rest of the industry.

The base alloy model presented here is a very abstracted dual clutch transmission. Things such as clutch and gear pairing configurations, vehicle speed and throttle thresholds, and other module interactions have not been added, but can be if further research is desired.

To summarize, this report finds that with a bit more work, Alloy as a model creator and checking tool for automotive transmissions is quite viable solution. From the cost aspect of the Alloy compared to other tools, as well as Alloy's capabilities, a replacement workflow could be offered to the automotive industry.

References

System for controlling vehicle automatic transmission using fuzzy logic **(US 5323318 A)**
https://www.google.com/patents/US5323318

Modeling an Automatic Transmission Controller
http://www.mathworks.com/help/simulink/examples/modeling-an-automatic-transmission-controller.html

YeD
http://www.yworks.com/en/products_yed_about.html