Validating Multiple Variants of an Automotive Light System with Electrum

Alcino Cunha, Nuno Macedo, and Chong Liu

INESC TEC & Universidade do Minho, Portugal

Abstract. This paper reports on the development and validation of a formal model for an automotive adaptive exterior lights system (ELS) with multiple variants in Electrum, a lightweight formal specification language that extends Alloy with mutable relations and temporal logic. We explore different strategies to address variability, one in pure Electrum and another through an annotative language extension. We then show how Electrum and its Analyzer can be used to validate systems of this nature, namely by checking that the reference scenarios are admissible, and to automatically verify whether the established requirements hold. A prototype was developed to translate the provided validation sequences into Electrum and back to further automate the validation process. The resulting ELS model was validated against the provided validation sequences and verified for most of requirements for all variants.

1 Introduction

Electrum [10] is a state-based modelling language that extends the structural definitions and first-order relational logic of Alloy [8] with mutable relations and (past and future) linear temporal logic (LTL) operators. Its companion Analyzer [2], itself an extension of the Alloy Analyzer, provides support for validation – through scenario animation – and verification – through two automatic model checking backends, one bounded and another complete. Both animation instances and verification counter-examples are presented back to the user in a unified graphical interface. The combination of first-order and temporal logic makes Electrum well-suited to address systems rich in both structural and dynamic properties, such as automotive software product lines with architectural and behavioural variability. To further ease the feature-oriented design of software families, language extensions to Alloy have also been proposed [1,9].

This paper reports the modelling and subsequent validation and verification of an *adaptive exterior lights* system (ELS) with multiple variants in Electrum[1], carried out as an answer to the ABZ'20 call for case study submissions, following the successful submission to ABZ'18 [4]. The employed approach – which we hope can be applied to similar signal-based systems – is presented in Section 2. As described in Section 3, we have been able to model most ELS requirements by

[1] All resources relevant for the ELS case study are available at `https://github.com/haslab/Electrum2/wiki/ELS`.

finding an abstraction sweet-spot – in particular for real-time issues. The Electrum language is presented throughout this section as needed. Section 4 describes two explored approaches to modelling multiple variants, one in pure Electrum and another using language extension for feature-oriented design [9]. The ELS model was validated against all the provided validation sequences [7], and verified for most of the ELS requirements, as described in Section 5. To ease validation, a prototype was developed to translate tabular validation sequences into Electrum and back for inspection by domain experts. Lastly, Section 6 discusses issues identified in the requirements and limitations of the followed approach.

2 Modelling strategy

The main goal of this work was to validate the ELS requirements by checking their feasibility and consistency for all valid variants. We started by modelling a single variant of the ELS as a (rough) state machine against which the validation sequences were tested and the requirements subsequently verified. An Electrum model contains both the system specification and the analysis commands, thus our model, described in detail in Section 3, is structured as follows:

Environment the available input and output signals, their acceptable values, and possible restrictions to their evolution [6, §4.1–4.3].

ELS state machine a predicate calculating the state of output signals (mostly) from the current state of the input ones, allowing alternative behaviours; inferred from the requirements [6, §4.4].

Animation scenarios simple state sequences, and associated run commands, that exercise the ELS for preliminary validation and regression testing.

Reference scenarios the encoding of the provided validation sequences [7], and associated run commands, with imposed inputs and expected outputs, for validating the modelled ELS state machine; a prototype was developed to translate them from the provided tabular format [7].

Visual elements elements ignored by the analyses but aiding the visualization of scenarios (accompanied by a theme, stored in a separate file).

Requirement assertions the formalization of the requirements [6, §4.4] in temporal logic, and associated check commands to automatically verify them; these assess the overall consistency of the ELS requirements.

The ELS has both structural – that introduce additional signals – and behavioural – that change certain signal outcomes – variability points [6, §3], which Electrum is well-suited to address. Once a single variant was modelled and validated, two strategies were explored to address the remaining variants, as described in Section 4: one based on an Electrum idiom where features and variability points are modelled in plain Electrum, and another adapting a language extension developed by us for Alloy [9] where variability points are annotated with features. This introduces another component in the ELS model for imposing the set of valid variants – the *feature model*.

As expected, the development of these components was not sequential but rather iterative as new ELS functions were added to the model. This process was applied to all 9 main ELS functions divided in 48 requirements as of version 1.17 [6], for all 12 valid variants (although only 4 effectively have distinct behaviour) and to all 9 validation sequences of version 1.7 [7]. This work focused on the ELS, but we believe a similar approach could be followed for the *speed control system* (SCS) [6], although the SCS is richer in continuous aspects, which would require additional abstractions. It should also be noted that the authors had no particular domain knowledge, and that the process was solely based on the provided reference material [6,7] and discussions with the case study chair.

The most challenging features of the ELS were those dealing with real-time aspects and the integer nature of the signals. For both we developed proper abstractions – respectively arbitrary duration events and value discretization, described in the next section – that still allowed us to address most requirements. Only requirements requiring arithmetic operations were not addressed at all.

3 The ELS Model

This section describes the main features of the ELS model developed for the simplest variant, that is, when the vehicle is not armoured and aimed at the EU market (the driver position does not affect the ELS).

System environment The ELS follows a typical architecture that communicates with the external world through input – from the user interface and sensors – and output signals – to actuators. Our model mimics this architecture so that the translation can be streamlined. In Electrum, likewise Alloy, structure is introduced through the declaration of *signatures* – sets of uninterpreted atoms – and *fields* declared within them – relations of arbitrary arity between signatures. A hierarchy on signatures can be imposed through simple *inclusion* or through *extension*, in which case children must be disjoint; signatures can also be declared as *abstract*, meaning all atoms must belong to its children. Signatures and fields can be restricted by simple *multiplicity* constraints. Lastly, both may be *static* (by default) or declared as *variable*, in which case their state may change over time.

The ELS environment specification declares signals, the values that can be assigned to such signals, and how these assignments are represented in time. Signals form a static hierarchy starting from an (abstract) `Signal` signature. Although the ELS signals are "flat", related signals are often better handled together. Thus, for instance, all light signals are aggregated in an abstract signature `Light`, and left and right low beam signals in the abstract signature `LowBeam`. At the bottom of the hierarchy are the concrete signals themselves as singleton signatures (multiplicity **`one`**), such as `LowBeamRight` and `LowBeamLeft`, whose names match those specified in the reference documents. The hierarchy relevant for the low beams function is encoded in Electrum as

```
abstract sig Light extends Signal { var state : one LightState }
abstract sig Beam, ... extends Light {}
```

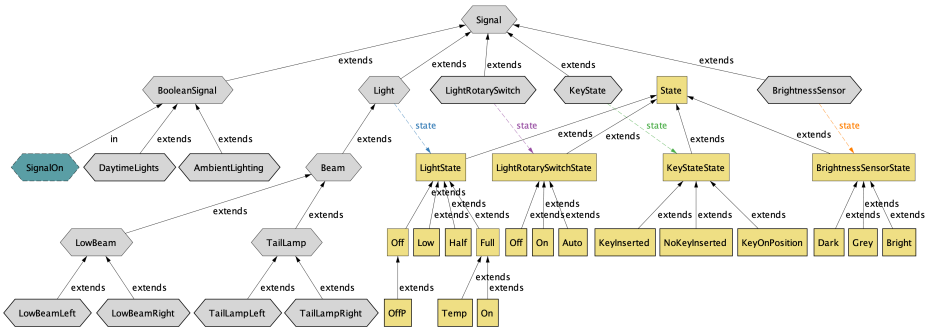

Fig. 1: Meta-model of the system environment model for low beam headlights.

```
abstract sig LowBeam, TailLamp extends Beam {}
one sig LowBeamLeft, LowBeamRight extends LowBeam {}
one sig TailLampLeft, TailLampRight extends TailLamp {}
```

where field `state` will be explained shortly. To simplify the modelling process, we distinguish Boolean signals (`BooleanSignal`, a sub-signature of `Signal`) from the others. Those relevant for the low beams function are declared as:

```
abstract sig BooleanSignal extends Signal {}
one sig AmbientLighting, DaytimeLights extends BooleanSignal {}
```

Although signals are integer, most requirements simply test whether they are within certain ranges. Thus, to keep the model manageable and avoid state explosion, we discretize the values of each signal into those ranges relevant for the requirements. For instance, it is only relevant to detect whether the ambient brightness levels are below 200, over 300 or between the two, while low beam headlights are only set to 20%, 50% or 100% intensity [6, 4.4]. Thus only these distinct classes of values are encoded in our model. Values form a hierarchy matching that of the signals, topped by `State`, whose direct children group the states of related signals, such as `LightState` for `Light` signals. The next layer provides the discretized values, such as `Off`, `Low`, `Half` or `Full` for beam intensity. Lastly, since our model abstracts real-time aspects, occasionally we require additional temporal context regarding the state of the signals. For instance, when low beams are activated due to ambient darkness, they must remain active for 3s even if ambient brightness is detected (ELS-18); thus, within `Full` beam intensity we distinguish between this temporary state (`Temp`) and permanent activation (`On`). Part of this hierarchy relevant for the low beam function is encoded as:

```
abstract sig LightState extends State {}
abstract sig Full, Off extends LightState {}
one sig Half, Low extends LightState {}
one sig On, Temp, ... extends Full {}
one sig OffP, ... extends Off {}
```

Lastly, we model the evolution of the state of the signals. For Boolean signals a variable sub-signature `SignalOn` will contain at each state all active signals:

```
var sig SignalOn in BooleanSignal {}
```

For the other signals, a variable field called `state` will contain at each state exactly **`one`** respective value, such as the one declared above for `Light` and respective `LightState`. Often the requirements impose certain restrictions on the evolution of the environment. In Electrum these restrictions are imposed through *facts*, representing model axioms, which can contain arbitrary temporal constraints. In the ELS, e.g., a fact forces the pitman arm to go back to neutral when the steering wheel returns to the vertical position [6, §4.1].

We have encoded the over 30 signals of the ELS in this manner, including those of the user interface [6, §4.1], the sensors [6, §4.2] and the actuators [6, §4.3]. An excerpt of the resulting environment signature hierarchy for the low beam headlights function is depicted in Fig. 1 as generated by the Analyzer. Dashed elements are variable and singleton signatures are in thicker lines (some state names are abbreviated). Throughout the rest of the paper we will rely on this function to demonstrate the features of the developed model.

State machine Next we derived a state machine from the ELS requirements. Electrum formulas are written in relational linear temporal logic with transitive closure. Relational expressions combine signatures and fields (and constants, namely the universe of atoms **`univ`**, the unary empty relation **`none`** and the identity relation **`iden`**) with typical set theory operators such as union (`+`), intersection (`&`), difference (`-`), Cartesian product (→), binary relation overriding (`++`), and relational join (`.`). In Electrum everything is seen as a relational expression, so `s.state` can be used to retrieve the current state of a concrete signal `s` or all the states of a set of signals `s`. Primed expressions can be used to refer to their value in the succeeding state, e.g., `s.state'` for the next value of `s.state`. Atomic formulas either test relational expressions for inclusion **`in`** or equality `=` or are simple multiplicity tests. So, `s in SignalOn` tests whether a Boolean signal `s` is currently active, and `s.state in v` whether signal `s` currently has value `v` (if `s` singleton). Complex formulas are composed by Boolean operators (e.g. **`not`**, **`and`**, **`or`**, **`iff`**, **`implies`** or **`implies`**-then-**`else`**), first-order operators (e.g. **`all`** or **`some`**), and future (unary **`after`**, **`always`** or **`eventually`**, or binary **`until`** or **`releases`**) and past (unary **`before`**, **`historically`** or **`once`**, or binary **`since`** or **`triggered`**) linear temporal logic operators. Predicates and functions can be defined for auxiliary formulas and expressions and let-expressions for local definitions.

A predicate is defined for each function encoding the expected behaviour, which are subsequently called in a fact that enforces the full state machine. For the low beam headlights function, this predicate mostly restricts the succeeding state of the low beam headlights given the current state of the other signals. For instance, if the light rotary switch (`LightRotarySwitch.state`) is set to `LSOn` while the key (`KeyState.state`) is in the ignition on position, the succeeding state of the low beams is set to `On` (ELS-14):

```
KeyState.state in KeyInIgnitionOnPosition and
  LightRotarySwitch.state in LSOn implies LowBeam.state' in On
```

Expression `LowBeam.state` aggregates the state of both the left and right low beams; since every light must have a `state` assigned, `LowBeam.state in On` sets both to full intensity. As a more complex example, consider ELS-17 that specifies daytime running lights, which activate the low beams when the engine is started until the key is removed from ignition, unless ambient light control is also active:

```
DaytimeLights in SignalOn and
  KeyState.state not in KeyInIgnitionOnPosition or
  (LowBeam.state in On and KeyState.state in KeyInserted and
    AmbientLighting not in SignalOn) implies LowBeam.state' in On
```

In our model, real-time is abstracted away and no particular duration is imposed to states, meaning that within a certain interval of time an arbitrary number of events may occur. This affects the modelling of events with a bounded duration, since we must identify when the trace is within that bound and allow multiple steps within. For this purpose, such events are explicitly identified in our state but not forced to last any particular number of states. For instance, the mandatory 3s for automatic low beams (ELS-18) is identified by the state `Temp`; when brightness is detected, the low beams may be turned `Off` or the `Temp` state propagated. This could be encoded in the following relational formula:

```
let low = LowBeam.state |
  LightRotarySwitch.state in LRSAuto and
    KeyState.state in KeyInIgnitionOnPosition implies
      one low' and
      BrightnessSensor.state in Dark implies
        low' in low.(univ→Temp+Temp→On++On→On) else
      BrightnessSensor.state in Bright implies
        low' in low.(univ→Off+Temp→Temp) else
      BrightnessSensor.state in Grey and low not in Temp implies
        low' in low.(iden+Temp→On)
```

Here `low` abbreviates the state of both left and right low beams and we rely on relational operators to specify alternative updates. For instance, expression `univ→Off+Temp→Temp` relates every state with `Off` and additionally `Temp` with itself; thus, `low.(univ→Off+Temp→Temp)` returns `Temp` and `Off` when the current state is `Temp` and solely `Off` otherwise. This allows the exploration of transitions with different durations: either low beams activation remains within the 3s, or the 3s are exceeded and they are deactivated. Formula `one low'` guarantees that left and right beams are updated consistently (i.e., with the same value). Liveness properties then guarantee that the system eventually evolves. In Electrum arbitrary temporal constraints can be imposed, this one taking the shape:

```
low in Temp implies eventually low not in Temp
```

This strategy was employed to model all the ELS main functions – direction blinking, hazard warning light, low beams, cornering lights, manual and adaptive high beams, emergency brake and reverse lights, and fault handling.

4 Handling Variability

The ELS assumes the existence of variability points, namely the market region, whether it is an armoured vehicle and the driver position (although this last does not affect the behaviour of the ELS) [6, §3]. The model described in the previous section represented a single ELS variant, and multiple independent models could be developed in such a way for each of the valid variants. However, such a strategy has poor maintainability and will not scale as the number of features increase. Electrum is sufficiently flexible to support systems with structural and behavioural variability points and effectively model families of software products. However, such idioms may be cumbersome, error-prone, and reduce comprehension, so to explore alternative approaches we implemented in Electrum an annotative language extension to natively support feature-oriented design. This extension was previously developed for Alloy but its adaptation to Electrum was straightforward. This section describes the design of the ELS family of products in both approaches, which allow simultaneously specifying and analysing all the 12 ELS variants. For both approaches, we assume the variant presented in the previous section to be the base variant, which is extended into a multi-variant model.

A pure Electrum idiom The first step in both approaches is to encode the feature model – the possible features and the constraints over them denoting the valid variants. When relying on a variability idiom, this is done by making features first-class elements of the model. A possibility is to create a signature (here, `Feature`) with an atom for each available feature (through singleton sub-signatures, such as `EU` or `ArmoredVehicle` for the ELS). A sub-signature then contains a particular selection of these features, representing the variant under analysis (here, `Variant`). Lastly, a fact restricts which variants are considered valid, in the case of ELS forcing a single market to be selected through a multiplicity test:

```
fact FeatureModel { one (EU+USA+Canada) & Variant }
```

To model architectural variability, conditional signatures and fields can be assigned a loose multiplicity that is restricted depending on the variant under analysis. In the ELS the darkness mode switch only exists on armoured vehicles, so its multiplicity is set to **`lone`** (at most one such signal exists), and then a fact forces its existence exactly when the respective feature is selected:

```
fact darknessModeSwitchOn {
  some DarknessModeSwitchOn iff ArmoredVehicle in Variant }
```

Behavioural variability can be modelled by testing which features are selected in `Variant` and adapting the relevant transitions of the state machine predicates. In the case of low beams, for instance, ambient lights should be ignored with active darkness mode in armoured vehicles (ELS-21), so the pre-condition for activating them when the engine is started (ELS-19) is adapted to:

```
not (ArmoredVehicle in Variant and DarknessModeSwitchOn in SignalOn)
  and AmbientLighting in SignalOn and BrightnessSensor.state in Dark
  and before KeyState.state in KeyInIgnitionOnPosition and
```

```
KeyState.state not in KeyInIgnitionOnPosition implies
  LowBeam.state' in Temp
```

Notice that since features are regular signatures, it may become difficult to identify which parts of the predicate are variability points. It may also led to unpredictable issues if the architectural variability is not handled with care: the distracted developer could simply write `DarknessModeSwitchOn in SignalOn` to test whether darkness mode is active without testing the feature presence, which is always true in variants without feature `ArmoredVehicle` since `DarknessModeSwitchOn` is empty, thus permanently disabling ambient lighting.

For an example regarding the USA and Canada market variants, during direction blinking, for instance, the intensity of daytime running lights (ELS-17) must be reduced to half in the respective side (ELS-6), so the transition shown in the previous section would be adapted to:

```
DaytimeLights in SignalOn and ... implies
  LowBeamLeft.state' in
    (some (USA+Canada) & Variant and BlinkLeft.state' not in OffP)
      implies Half else On and
  LowBeamRight.state' in
    (some (USA+Canada) & Variant and BlinkRight.state' not in OffP)
      implies Half else On
```

where the state of the blinking lights `BlinkLeft` and `BlinkRight` is tested in case the USA or Canada markets are selected.

A colourful Electrum extension Approaches to explicitly introduce variability in a system usually fall in two categories: *compositional* approaches where features are implemented as distinct code units which are then composed when creating a variant, and *annotative* approaches where the code is annotated to dictate which fragments will appear in each variant. Both compositional [1] and annotative [9] approaches have been proposed to enable feature-oriented design in Alloy, the latter by us relying on colourful annotations that have been shown to improve understandability [5]. Annotative approaches are better suited for small granularity variability points, which in our experience is often the case in Alloy/Electrum, such as the examples above where one needs to change part of a formula or expression rather than replace the predicate altogether.

In our lightweight annotative approach model elements can be marked with features, identified by a digit, to control their presence/absence without obfuscating the code. Positive and negative annotations are introduced, respectively, by delimiters ⓘ and ❶ for $1 \leq i \leq 9$, and colour highlighted by the Analyzer. These can be nested, representing the conjunction of presence conditions, and be applied to most declarations or branches of certain operators (namely conjunction, disjunction, intersection and union). Semantically, when the presence conditions are not met the element is interpreted as the neutral element of the respective operator. For instance, in ⓵*p*⓵ **and** ❷*q*❷, *p* is only tested in variants with feature 1, and *q* in those without feature 2, being replaced by *true* otherwise.

The multi-variant ELS model under this extension uses five feature annotations, one for each variability point. To model the feature model one can rely on

annotated facts to forbid certain variants. For the ELS this could be achieved by the following fact, which mimics the colour highlighting of the Analyzer:

```
fact FeatureModel {
  // ① USA, ② Canada, ③ EU, ④ Armored, ⑤ DriverPosition
  ①②some none②① and ②③some none③②
  ①③some none③① and ❶❷❸some none❸❷❶ }
```

where, for instance, ①②**some none**②① forbids the coexistence of USA and Canada market codes, and ❶❷❸**some none**❸❷❶ forces the selection of at least one market code[2]. At the level of abstraction of Electrum, feature models are usually small and simple to encode with facts like the one above, but we are studying whether dedicated support for encoding feature models is necessary.

Architectural variability is trivially modelled, as one may mark the signature (or field) declaration with the relevant annotations, as in the case of the darkness mode switch signal,that only exists for armoured vehicles:

```
④one sig DarknessModeSwitchOn extends BooleanSignal ④
```

One type rule imposed by colourful Electrum is that element calls must respect the annotations in which they were declared, thus guaranteeing that they are never called in variants where the element is absent. Thus, the interaction between ELS-19 and ELS-21 would now be encoded as:

```
④not DarknessModeSwitchOn in SignalOn④ and
AmbientLighting in SignalOn and BrightnessSensor.state in Dark and
  ... implies LowBeam.state' in Temp
```

In variants without ④ this test will be disregarded (i.e., interpreted as *true*). The same mechanism can be applied to relational expressions. For instance, the interaction of ELS-17 and ELS-6 for USA and Canada markets is encoded as:

```
DaytimeLights in SignalOn and ... implies
  LowBeamLeft.state' in
    ③On③+❸BlinkLeft.state' not in OffP implies Half else On❸ and
  LowBeamRight.state' in
    ③On③+❸BlinkRight.state' not in OffP implies Half else On❸
```

where the beams are always set to `On` in the EU market, but in other markets (through the negative ❸) the state of blinking lights is tested. A union branch is interpreted as the empty relation when the presence conditions do not hold.

5 Validation & Verification

The Analyzer is able to execute animation and verification commands. Both instances and counter-examples are graphically depicted in a visualizer that can be customized for improved interpretation. This section describes how these functionalities were used to validate and verify the ELS model.

[2] Electrum, like Alloy, does not natively support Boolean constants, so **some none** is commonly used to denote a trivially unsatisfiable formula.

5.1 Animation and Validation

Validation scenarios Animation commands are defined through **run** instructions, which can be provided arbitrary constraints that must hold for the generated instances. This allows the quick definition of scenarios for early validation, which are also useful as regression tests as the model evolves. For the ELS we have defined over 60 such scenarios exercising simple behaviours of the system. We follow an idiom where one predicate defines the evolution of the environment (state of input signals) and another the expected behaviour of the system (state of output signals). For instance, to test basic low beam headlights sub-functions such as having the light rotary switch set to on with key inserted, a predicate is defined to encode the behaviour of the relevant input signals:

```
pred LowBeam2Env {
  always AmbientLighting not in SignalOn
  always KeyState.state in KeyInserted
  let lrs = LightRotarySwitch.state |
   lrs in LSOff;always lrs in LSOn }
```

where **always** *p* forces *p* to hold in all states of the trace and *p*;*q* abbreviates *p* **and after** *q*, an operator introduced precisely to ease scenario specification [4]. A predicate then encodes the expected outcome of the ELS for these inputs:

```
pred LowBeam2Exp {
  LowBeam.state in OffP;always LowBeam.state in Half }
```

This predicate states that the beams should be activated with intensity reduced to half. Lastly, a command to generate this scenario by enforcing the environment and the expected behaviour (in the succeeding state, since output signals are calculated from the previous state) is defined:

```
run LowBeam2 { LowBeam2Env and after LowBeam2Exp } for 5 Time
```

Commands must have scopes assigned to signatures, but in our ELS model all signatures are exactly bound, since all signals and possible states are known a priori. For bounded model checking – more efficient and thus better suited for validation – the maximum number of states that form a trace must also be provided (the scope of **Time**). Since this is a simple scenario that bound is set to 5. Once instances are generated, the user is able to iterate over alternative scenarios for which the constraints hold. Scenario exploration operations (see the toolbar of Fig. 2) include changing the configuration (here, the selected variant), the initial state, or the current transition [3].

In the multi-variant ELS models one is able to restrict which subset of variants should be analysed. As an example, let us consider the animation of the effect of darkness mode when ambient lighting is activated. In the Electrum variability idiom the part of this environment predicate could be specified as:

```
ArmoredVehicle in Variant
let key = KeyState.state |
  key in KeyInIgnitionOnPosition;always key in KeyInserted
always AmbientLighting in SignalOn
always DarknessModeSwitchOn in SignalOn
```

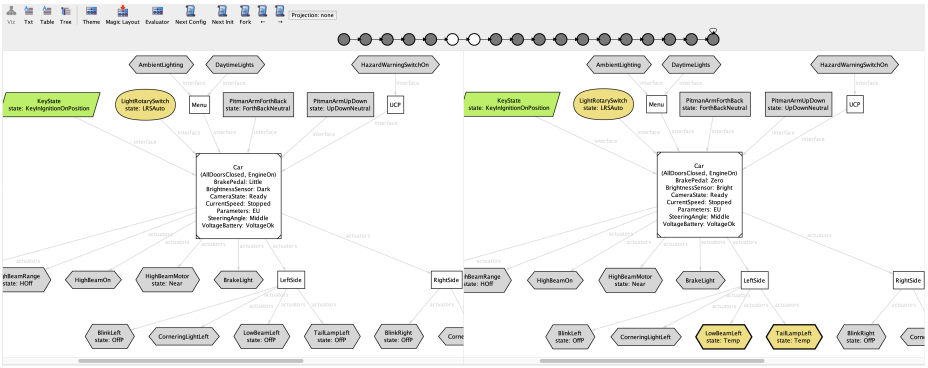

Fig. 2: A step of sequence 1 in the Analyzer under the developed theme.

which includes the selection of the feature `ArmoredVehicle` and the behaviour of the `DarknessModeSwitchOn`. The same scenario in the colourful extension would instead be specified as:

```
let key = KeyState.state |
  key in KeyInIgnitionOnPosition;always key in KeyInserted
always AmbientLighting in SignalOn
④always DarknessModeSwitchOn in SignalOn④
```

where the behaviour of the darkness mode switch is annotated with the corresponding feature. The execution of this scenario must then also be restricted to only variants where feature 4 is selected. In colourful `Electrum` this is defined through the command scope as:

```
run LowBeam19 {
  LowBeam19Env and after LowBeam19Exp } with ④ for 5 Time
```

Theme customizations In our experience, the proper graphical representation of instances is key to promote the interpretation of the model among interested parties. Inheriting from Alloy, the Analyzer depicts instances as graphs, applying a graph representation algorithm and distributing nodes among layers, obliviously of the underlying semantics of the nodes and edges. Themes may be defined to ease interpretation. From our experience the most useful customizations are simply changing the colour, shape or label of elements, hiding elements, showing relations as edges or attributes, and inverting edges (the easiest way to change the shape of the graph). Visualization can also be projected over a signature, focusing the visualization on the elements related to the selected atom. These customizations are hierarchical, meaning that subsets of elements may inherit the parameters of their parents or change them. Although simple, these features can become extremely powerful given another key functionality of the visualizer – after analysis, and during the creation of the graph, auxiliary functions defined in the model are introduced into the instance. These can be of arbitrary arity, and thus can represent subsets of atoms or new relations between them.

In our ELS model we have used such features to produce a visualization such as that of Fig. 2. Since the signals are mostly flat, we introduce elements to somehow layout signals according to their role in the system. Singleton signatures – which do not affect the solving process since they are exactly bound and not referred elsewhere – simulate the vehicle architecture, such as the `Car` itself or the driver's `Menu`:

```
one sig Car, LeftSide, RightSide, Menu, UCP {}
```

Auxiliary relations (defined as functions with zero arguments) then connect such elements to signals, such as assigning the sensors to the car (which are set to be shown as attributes of `Car` rather than edges) or the lights to the respective side of the car, and can be defined as follows:

```
fun _lightsensor : Car → BrightnessSensorState {
  Car → BrightnessSensor.state }
fun _actuators : univ → univ {
  LeftSide → (BlinkLeft+LowBeamLeft+TailLampLeft+...) +
  RightSide → (BlinkRight+LowBeamRight+TailLampRight+...) }
```

Auxiliary sets grouping together signals under certain states were also defined to ease the theme customization. For instance, all active signals are grouped so that they can easily be painted with a distinguishing colour (yellow in Fig. 2):

```
fun _on : set univ { state.Full+state.(LSOn+LSAuto)+SignalOn+... }
```

The theme file is available alongside the model specification.

5.2 Reference validation sequences

To effectively validate the developed model we checked its behaviour against that of the reference validation sequences [7]. These are complex – each step specifying the value of all the over 30 input and output signals, with some containing over 20 steps – rendering their manual codification infeasible. Thus, we implemented a prototype to automatically translate tabular data that represents signal values over time into Electrum and back. This validator is able to *i*) given a sequence of input and output signals, report whether it is a valid execution in our model; and *ii*) given a sequence of only input signals, generate possible executions of the output signals to be subsequently validated by domain experts.

We implemented the prototype so that the process could be reproducible for other signal-based systems. Thus, besides the sensor data, two additional pieces of information must be provided to the validator for each specific application: *i*) how the signal values should be discretized; and *ii*) the presence conditions for signals. For our prototype, this information is passed in the header of the tabular data, as depicted in Table 1 for validation sequence 1 of the ELS (note that this is only an excerpt of the codification of the more than 30 signals over 17 steps). Single-value ranges are assumed to have the same lower- and upper-bound. It also assumes, as described in Section 3, that all signals are leaves of the hierarchy on `Signal` with the exact same name as that of the sequence header, and that

Table 1: Snippet of tabular data provided to our validator for sequence 1.

...	Time	ambient Lighting	darknessMode SwitchOn	lightRotary Switch	brightnessSensor	marketCode	armored Vehicle	...	lowBeam Left	...
...		0=False; 1=True	0=False; 1=True	0=Off; 1=Auto; 2=On	0-199=Dark; 200-250=Grey; 251-100000=Bright	1=USA; 2=Canada; 3=EU	1=True; 0=False	...	0=Off; 10=Low; 50=Half; 100=Full	...
...			armored Vehicle=True					...		...
...	...	...	...	...	...	...	...	...	...	...
...	0:03	0	0	1	500	3	0	...	0	...
...	0:04	0	0	1	200	3	0	...	0	...
...	0:05	0	0	1	199	3	0	...	100	...
...	...	...	...	...	...	...	...	...	...	...

```
1  let s1 = not AmbientLighting in SignalOn |
2    always s1
3  let s1 = not DarknessModeSwitchOn in SignalOn |
4    always s1
5  let s1 = LightRotarySwitch.state in LSAuto, s0 = LightRotarySwitch.state in LSOn,
6      s2 = LightRotarySwitch.state in LSOff |
7    s2;s2;s2;s1;s1;s1;s1;s1;s1;s1;s0;s0;s1;s0;s0;s0;always s1
8  let s0 = BrightnessSensor.state in Dark, s1 = BrightnessSensor.state in Grey,
9      s2 = BrightnessSensor.state in Bright |
10   s2;s2;s2;s2;s1;s0;s2;s0;always s2
11 EU in Variant
12 ArmoredVehicle not in Variant
13 ...
14 after {
15   let s2 = LowBeamLeft.state in LightLow, s3 = LowBeamLeft.state in LightOff,
16       s1 = LowBeamLeft.state in LightHalf, s0 = LowBeamLeft.state in LightFull |
17     s3;s3;s3;s3;s3;s0;s0;s0;s3;s3;s0;s3;s3;s1;s3;s2;always s3
18   ... }
```

Fig. 3: Electrum encoding of the sequence from Table 1.

elements representing the discretized values are at the second layer of the `State` hierarchy, again with the same name as the discretization in the header.

The translation can then be streamlined as follows. The presence/absence of a Boolean signal *s* can simply be stated as *s* **in** `SignalOn` and *s* **not in** `SignalOn`, respectively, while the state of the others is encoded as *s*`.state` **in** *v* for a discretized value *v*. Sequences of signal states are encoded using the operator `;`, and let-expressions are used to simplify this codification. The particular variant of the sequence must also be encoded. The validator currently implements only the pure variability idiom, forcing the exact value of signature `Variant`.

The resulting predicates resemble the one in Fig. 3 for the sequence from Table 1 (including steps that have been omitted for simplicity). The expected variant (ll.11–12) and both the sequence of input (ll. 1–10) and output (ll. 15–17) signals are encoded, relying on let-expressions for improved readability (recall that unlike the validation sequences, our output signals are only updated in the succeeding state, hence the **`after`**). At the last state an **`always`** operator is applied, since outputs are expected to stabilize when inputs do. Although the

reference sequences provide timestamps for the events (the first column), these are ignored since real-time is abstracted in our model.

Figure 2 depicts the outcome of running this predicate (with **Time** scope determined from the length of the sequence), particularly the transition where the brightness is below the threshold and the low beam headlights are activated. We were able to model all 9 validation sequences of version 1.8 and show that they hold for our ELS model, except for concrete values for the high beam illumination distance and strength in sequence 9 (ELS-33) due to arithmetic operations.

5.3 Requirement verification

The last step of the process was to effectively verify whether the requirements hold for the modelled ELS. In Electrum assertions (**assert**) can be specified in full relational temporal logic, which the Analyzer is instructed to verify (within given scopes) with **check** commands.

As an example, consider requirement ELS-14, stating that whenever the engine is on and the light switch set to on, low beams will be active. This can be specified in the following temporal assertion:

```
assert ELS14 {
  always (KeyState.state in KeyInIgnitionOnPosition and
    LightRotarySwitch.state in LSOn implies LowBeam.state' in Full) }
```

For a more complex example, consider ELS-17, stating that with daytime running light but without ambient light, the low beams are activated until the engine is turned off. This can be encoded as:

```
assert ELS17 {
  let keyPos = KeyState.state in KeyInIgnitionOnPosition,
      amb = AmbientLighting in SignalOn,
      day = DaytimeLights in SignalOn |
  always (day and not amb) implies always (
    (LowBeam.state' in Full+Half until not keyPos) or always keyPos) }
```

stating that in traces where daytime running light is active but not ambient lighting, the engine is turned off and the low beams are deactivated (temporal operator **until**) or the engine remains on forever.

We were able to check most ELS requirements except for the limitations discussed in the following section. The described checks (that verify the property for all variants at once) take around 6s and 10s, respectively, using the bounded engine of Electrum under the Glucose SAT solver and for 15 **Time** in a commodity 2,3 GHz Intel Core i5 with 16GB RAM. More complex requirements – like those including periodic events such as ELS-2 and ELS-4 – take around 1min.

6 Results Discussion

The reference document Throughout the development of the ELS model we encountered 14 issues with the reference documents, mostly during modelling and

preliminary validation, and when running the reference sequences. We reported them to the case study chair who promptly replied. Of the first 4 reported issues, 3 resulted in fixes to the reference document (version 1.11); unfortunately, at the time of submission no new version has been released after the other interactions (unofficially, at least 3 resulted in validation sequence fixes). Roughly, the issues encountered were either with the

environment model inconsistencies or missing features related to the signals detected in the early modelling process (e.g., the lack of a signal for the middle brake light, making it impossible to flash (ELS-40); or inconsistent representations of the pitman arm signals when it was split into two distinct signals for vertical and horizontal movement);

behavioural model ambiguities detected in the requirements while modelling and animating the state machine (e.g., conflicting requirements where the precedence is not explicitly stated, such as whether ELS-18 or ELS-19 has priority on low beam behaviour; ambiguous nomenclature, such as what activating high beams means for the 3 relevant signals; or under-specified behaviour, such as the beam intensity of tail lamps);

validation sequences inadmissible sequences, meaning that the expected output signals could not be achieved from the input signals in our model (e.g., tail lamps not being activated or not blinking in sequence 7).

It must also be noted that, since the modelling and validation process was iterative, some requirement ambiguities were clarified by observing the reference sequences. For instance, it is not clear from ELS-22 that when tail lamps are activated, they are so with the same intensity as that of the low beams, but the sequences showed that to be the case (e.g., in ELS-15).

In our experience, there were two main sources of confusion in the requirements. One has to do with the blinking lights and the nature of the dark cycles: it was not clear under which situations, if any, such cycles should be interrupted, and under which situations do they impact the tail lamps. The second has to do with high beam headlights, which are controlled by 3 distinct signals: it is often not clear what it means to activate the high beams and how the 3 signals should be updated and again how they relate to the intensity of the tail lamps.

The followed approach As already stated, we only failed to address requirements requiring arithmetic operations (ELS-33 for calculating the illumination distance and luminous strength of high beams, and ELS-47 for calculating the maximum light intensity under over-voltage) since concrete integer values are not represented. The abstracted time also renders reasoning about real-time requirements infeasible, such as ELS-10 enforcing the duration of blinking cycles to 1s, or the part of ELS-18 forcing the activation of the automatic low beams for 3s. Some features were simplified to avoid additional internal states, namely the gentle fade-out of cornering lights (ELS-24) or the flashing of emergency brake lights (ELS-40). ELS-37, dealing with the interaction with the SCS, has been disregarded. Requirements related to periodic events – such as the bright and dark cycles of blinking lights – proved to be the most cumbersome to specify.

The multiple variants of the ELS requirements motivated the implementation of the feature annotations for Electrum and its Analyzer. Since the ELS is not particularly rich in variability, we did not find multi-variant modelling in a pure Electrum idiom to be unmanageable, but it did affect the comprehension of the model. In general, the colourful Electrum model is easier to understand. The exception is the axiomatization of the feature model, and we are already studying sensible ways to improve it, that we also expect to be useful in more advanced feature-oriented analysis procedures. The complexity of the case study also helped us identify additional operators whose annotation would be useful in colourful Electrum – namely, if-then-else expressions common in the definition of state machines, when certain branches are only relevant in certain variants.

Acknowledgements The authors would like to thank Frank Houdek for helping clarifying the requirements. This work is financed by National Funds through the Portuguese funding agency, FCT – Fundação para a Ciência e a Tecnologia, within project UIDB/50014/2020. The third author was financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation – COMPETE 2020 Programme and by National Funds through the FCT, within project POCI-01-0145-FEDER-016826.

References

1. Apel, S., Scholz, W., Lengauer, C., Kästner, C.: Detecting dependences and interactions in feature-oriented design. In: ISSRE. pp. 161–170. IEEE (2010)
2. Brunel, J., Chemouil, D., Cunha, A., Macedo, N.: The Electrum Analyzer: Model checking relational first-order temporal specifications. In: ASE. pp. 884–887. ACM (2018)
3. Brunel, J., Chemouil, D., Cunha, A., Macedo, N.: Simulation under arbitrary temporal logic constraints. In: F-IDE@FM. EPTCS, vol. 310, pp. 63–69 (2019)
4. Cunha, A., Macedo, N.: Validating the Hybrid ERTMS/ETCS Level 3 concept with Electrum. International Journal on Software Tools for Technology Transfer (2019)
5. Feigenspan, J., Kästner, C., Apel, S., Liebig, J., Schulze, M., Dachselt, R., Papendieck, M., Leich, T., Saake, G.: Do background colors improve program comprehension in the #ifdef hell? Empirical Software Engineering 18(4), 699–745 (2013)
6. Houdek, F., Raschke, A.: Adaptive exterior light and speed control system (2019), v1.17
7. Houdek, F., Raschke, A.: Validation sequences for ABZ case study "adaptive exterior light and speed control system" (2019), v1.8
8. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, revised edn. (2012)
9. Liu, C., Macedo, N., Cunha, A.: Simplifying the analysis of software design variants with a colorful Alloy. In: SETTA. LNCS, vol. 11951, pp. 38–55. Springer (2019)
10. Macedo, N., Brunel, J., Chemouil, D., Cunha, A., Kuperberg, D.: Lightweight specification and analysis of dynamic systems with rich configurations. In: SIGSOFT FSE. pp. 373–383. ACM (2016)