Noname manuscript No.
(will be inserted by the editor)

Validating Multiple Variants of an Automotive Light System with Alloy 6

Alcino Cunha · Nuno Macedo · Chong Liu

the date of receipt and acceptance should be inserted later

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 Alloy 6. Alloy 6 is the most recent version of the Alloy lightweight formal specification language that supports mutable relations and temporal logic. We explore different strategies to address variability, one in pure Alloy and another through an annotative language extension. We then show how Alloy 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 Alloy 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.

Keywords Formal specification · Model checking · Feature-oriented design

1 Introduction

Alloy [16] is a lightweight formal specification language based on relational logic. Its most recent version, Alloy 6 (previously known as Electrum), besides allowing structural definitions and constraints in first-order relational logic, also supports mutable relations and (past and future) linear temporal logic (LTL) constraints [22]. Its companion Analyzer [5] provides support for validation – through scenario animation – and verification – through two automatic model checking backends, one bounded and another complete [21]. 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 Alloy 6 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, composition- [1] and annotatation-based [20] extensions to previous versions of Alloy have also been proposed.

This paper reports the modelling and subsequent validation and verification of an *adaptive exterior lights* system (ELS) with multiple variants in Alloy 6, carried out as an answer to the ABZ'20 call for case study submissions [15], following our successful submission to the ABZ'18 case study [8]. The employed approach – which we hope can be applied to similar signal-based systems – is presented in Section 2, and the resulting model described in Section 3. As described in Section 3.2, we have been able to model most ELS requirements by finding proper abstractions, in particular for quantitative and real-time issues. The Alloy 6 language is presented throughout this section as needed. Section 3.3 describes two explored approaches to modelling multiple variants, one in pure Alloy 6 and another using a language extension for feature-oriented design [20]. The ELS model was validated against all the provided validation sequences [14], and verified for most of the ELS requirements, as described in Section 4. To ease validation, a prototype was developed to translate tabular validation sequences into Alloy 6 and back for inspection

A. Cunha, N. Macedo and C. Liu
INESC TEC, Porto, Portugal

A. Cunha and C. Liu
Universidade do Minho, Braga, Portugal

N. Macedo
Faculdade de Engenharia da Universidade do Porto, Porto, Portugal

by domain experts. Section 5 discusses issues identified in the requirements and limitations of the followed approach. Section 6 compares our approach with other submissions to the case study call, while Section 7 draws conclusions and wraps up the paper.

The team has extensive background on using, teaching, and performing research on topics related with lightweight formal methods, and Alloy/Electrum in particular. The main modelling activities were carried out by Macedo, double-checked by Cunha. Liu focused on variability modelling. In Alloy, the design philosophy is to the model, properties and commands in the same `.als` file. The annotative extension follows this philosophy, with an `.als` file containing multiple variants and their dependencies. These files (along with `.thm` files for theme customization) were synchronized among the team through a git repository[1]. 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 [13,14] and discussions with the case study chair.

This paper extends a conference version [9] by detailing the modelling strategy and the resulting model (in particular regarding the evolution of the system's environment and the abstraction of timing aspects), further discussing the limitations of the proposed approach, and providing a comparison with the other approaches to proposed to address the ELS case study [3,18,19,23,24] (Section 6). At the time of the original submission, Electrum was an independent extension of Alloy, but it has since been integrated into the official Alloy 6 release. The text has also been adapted to reflect this fact.

2 Requirements and Modelling strategy

The main goal of this work was to validate the ELS requirements for all valid variants by exploring possible designs, thus checking the feasibility and consistency of the requirements. We started by modelling a single variant of the ELS as a (loosely specified) state machine against which the validation sequences were tested and the requirements subsequently verified. An Alloy model contains both the system specification and the analysis commands, thus our model, described in detail in Section 3.2, is structured as follows:

Environment the available input and output signals, their acceptable values, and possible restrictions to their behaviour [13, §4.1–4.3]. Traceability to the reference document is kept via the naming of each signature associated with a signal.

ELS state machine a predicate calculating the state of output signals (mostly) from the current state of the input ones, allowing alternative behaviours; it has been inferred from the reference material, particularly from the requirements [13, §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 [14], and associated run commands, with imposed inputs and expected outputs, for validating the modelled ELS state machine. A prototype was developed to translate sequences from the provided tabular format [14] into the model. Traceability to the reference document is kept via the naming of each predicate/run command associated with a validation sequence.

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

Requirement assertions the formalization of the requirements [13, §4.4] in temporal logic, and associated check commands to automatically verify them. These assess whether the expected requirements hold for the design ELS. Traceability to the reference document is kept via the naming of each assertion/check command associated with a requirement.

Once a single variant was modelled and validated, we tackled the variability of the ELS. The first step was to introduce an additional model element:

Feature model introduces the available features and restricts valid configurations, as stated in the reference document [13, §3]. Traceability to the reference document is documented as comments in the declaration of the feature model.

The ELS has both structural – that introduce additional signals – and behavioural – that change certain signal outcomes – variability points that depend on the selected features. The Alloy 6 language is flexible enough to support such alternative requirements, and all the model elements described above were adapted to support variability.

We explored and compared two distinct strategies to encode the feature model and the variability points, which are detailed in Section 3.3:

– an approach based on an Alloy 6 idiom, where features are introduced as optional Alloy elements and variability points as conditionals over their presence, a strategy known as configuration lifting [25] or variability encoding [2];
– an annotation-based approach to feature-oriented design through an extension developed by us for Alloy

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

5 [20], that we believe is suited for the Alloy level of abstraction where variability points are fine-grained (in contrast to composition-based approaches [17], that require each feature to be developed in a distinct module).

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 [13], for all 12 valid variants (although only 4 effectively have distinct behaviour) and to all 9 validation sequences of version 1.7 [14]. This work focused on the ELS, but we believe a similar approach could be followed for the *speed control system* (SCS) [13], although the SCS is richer in continuous aspects, which would require additional abstractions.

The features of the ELS most challenging to model in Alloy were those dealing with real-time aspects and the integer nature of the signals. Two main abstractions were introduced in our model to address these two issues, respectively arbitrary duration events and value discretization, described in the next section. This still allowed us to address most requirements. Only requirements effectively requiring arithmetic operations were not addressed at all.

3 Model details

3.1 Alloy in a Nutshell

In Alloy 6, likewise previous versions, structure is introduced through the declaration of *signatures* (keyword **`sig`**) – 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* (**`in`**) or through *extension* (**`extends`**), 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, such as **`some`** (there is at least some element), **`lone`** (there is at most one element), or **`one`** (there is exactly one element). Lastly, both may be *static* (by default) or declared as *mutable* (**`var`**), in which case their state may change over time.

Alloy constraints are based on *temporal relational logic*, an extension of first-order logic with transitive closure and past and future linear temporal logic operators. 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 (`.`). To simplify syntax and semantics, in Alloy everything is seen as a relation: signatures are sets (unary relations), while scalars and quantified variables are singleton sets. So besides being used to compose relations, the join operator also subsumes relation application. Primed expressions can be used to refer to their value in the succeeding state.

Atomic formulas either test relational expressions for inclusion (**`in`**) or equality (`=`), or are simple multiplicity tests. 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.

An additional restriction over the model is imposed through a **`fact`**. These represent model axioms, which can contain arbitrary temporal relational formulas. In contrast, a property that is expected to hold in the model as a consequence of the imposed facts is defined as an **`assert`**. *Predicates* (**`pred`**) and *functions* (**`fun`**) can be defined for auxiliary formulas and expressions, respectively, and **`let`**-expressions for local definitions.

Lastly, an `Alloy` model can contain *commands* to be executed by the Analyzer. Animation instructions to generate model instances are defined through **`run`** commands, which can be provided arbitrary constraints that must hold for the generated instances. Instructions to be verify assertions are given as **`check`** commands. These commands always consider a bounded universe, which are controlled by assigning *scopes* (keyword **`for`**) to the declared signatures.

3.2 The ELS Model

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

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 into Alloy can be streamlined.

The ELS environment model 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 simply integers

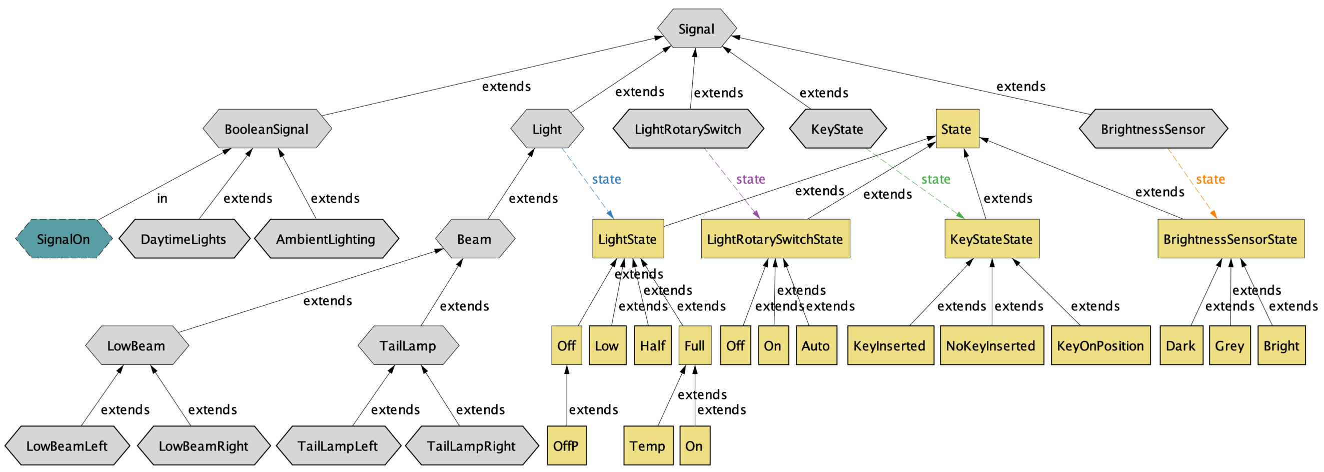

Fig. 1: Meta-model of the system environment model for low beam headlights.

without any additional structure, grouping together related signals in a common signature eases modelling and comprehension. 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 Alloy as:

```
abstract sig Light extends Signal {
  var state : one LightState }
abstract sig Beam, ... extends Light {}
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. Boolean signals relevant for the low beams function are declared as:

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

Although signal values are integer numbers, 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 [13, §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 (this process is detailed shortly when presenting the state machine). 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 mutable sub-signature `SignalOn` will contain at each state all active signals:

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

So, if *s* is a single Boolean signal, *s* **in** `SignalOn` tests whether *s* is currently active. For the other signals, a mutable field called `state` will contain at each state exactly **one** respective value, such as the one declared above for `Light` pointing to a `LightState`[2]. Expression

[2] To simplify the model, we could push the `state` fields to an abstract signature which all signatures with mutable state would extend. However, this would have a heavy toll on performance of the Alloy solving procedures, since such `state` would have the general type `State` even for concrete signals.

`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`. If `v` is some single value, `s.state in v` tests whether all signals in `s` currently have value `v`. A primed expression `s.state'` retrieves the next value of `s.state`.

We have encoded the over 30 signals of the ELS in this manner, including those of the user interface [13, §4.1], the sensors [13, §4.2] and the actuators [13, §4.3]. The Alloy meta-model of the environment signature hierarchy for the low beam headlights function is depicted in Fig. 1, as generated by the Analyzer. Signals are represented as grey hexagons and signal states as yellow boxes; singleton signatures, the leaves of the hierarchy, are depicted in thicker lines. Mutable elements are drawn with dashed lines, namely the (blue) subset of active signals and the state assigned to each signal[3]. Instances of the Alloy model will follow a similar style, albeit with additional customizations. Throughout the rest of the paper we will mostly rely on this function, the low beam headlights, to demonstrate the features of the developed models.

Often, there are some assumptions on the evolution of the system's environment, which can be controlled through facts. In the ELS model, a fact forces the pitman arm (`PitmanArmUpDown.signal`) to go back to neutral when the steering wheel (`SteeringAngle.signal`) returns to the vertical position from another state [13, §4.1]:

```
fact Environment { always (
  (PitmanArmUpDown.state in Downward7+Upward7 and
  SteeringAngle.state in Left+Right and
  SteeringAngle.state' in Middle) implies
    PitmanArmUpDown.state' in UpDownNeutral) }
```

State machine Next we derived a state machine from the ELS requirements. A predicate is defined to encode the behaviour of each function, 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 signals 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 KeyOnPosition 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 beams to full intensity. As a more complex example, consider ELS-17 that specifies daytime running lights, which states the low beams are activated when the engine is started and remain so until the key is removed from ignition, unless ambient light control is also active:

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

This formula forces the state of the low beams to be `On` in the next state if `DaytimeLights` is active and, either the engine is started (`KeyOnPosition`), or the key is in the ignition (`KeyInserted`) and the low beams were already `On` (as long as `AmbientLighting` is not active, a case handled instead by ELS-19).

In our model, real-time is abstracted away and no particular duration is imposed to states. So, an ELS event with a bounded duration – such as low beam headlights always remaining active for at least 3s (ELS-18) – may take an arbitrary number of steps to terminate in our model. The allows the exploration of simultaneously occurring events with alternative interleavings. To that purpose, the model's state registers when events with a bounded duration are executing, and at each state they are either allowed to continue executing or terminate; an additional constraint forces them to inevitably terminate at some point. For instance, the mandatory 3s for automatic low beams (ELS-18) is identified by the headlight state `Temp`; when brightness is detected, the low beams may be turned `Off` or the `Temp` state propagated to model the possible delay to complete the 3s. This could be encoded in the following relational formula:

```
let low = LowBeam.state |
(LightRotarySwitch.state in LRSAuto and
KeyState.state in KeyOnPosition) 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 beam headlights and we rely on relational expressions to specify alternative updates. For instance, binary expression `univ→Off+Temp→Temp` relates every state value with `Off` (`univ→Off`) and additionally state value `Temp` with itself (`Temp→Temp`); thus, `low.(univ→Off+Temp→Temp)` returns `Temp` and `Off` when the current state is `Temp` and only `Off` otherwise. As mentioned above, this allows events to last an arbitrary number of states, which are not given any real-time interpretation: whenever low

[3] Some state names are abbreviated in the paper to ease presentation but are consistent with the reference document in the full model, such as `KeyOnPosition` rather than as `KeyInIgnitionOnPosition`.

beams have state value `Temp`, either the 3s have not elapsed and the low beams remain active (`Temp` is selected again), or the 3s have been exceeded and the low beams are deactivated (`Off` is selected instead). Formula **`one`** `low'` guarantees that left and right beams are updated consistently (i.e., with the same state value). Liveness properties then guarantee that the system eventually evolves, since in Alloy 6 arbitrary temporal constraints can be imposed. For ELS-18 this formula takes the following shape, forbidding the low beams to stay in state `Temp` indefinitely (i.e., the 3s eventually elapse):

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

The strategy described throughout this section 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.

3.3 Handling Variability

The ELS has variability points, namely the market region, whether it is an armoured vehicle and the driver position (although this last one does not affect the behaviour of the ELS) [13, §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. Alloy 6 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 previously implemented in Alloy an annotative language extension to natively support feature-oriented design. This extension was developed for Alloy 5 but its adaptation to Alloy 6 to model the ELS 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 Alloy 6 idiom The first step in both approaches is to encode the feature model – the possible features and the constraints over them, representing the set of valid variants. When relying on a variability idiom, this is done by making features first-class elements of the model over which the presence conditions can be tested (a strategy known in the community as configuration lifting [25] or variability encoding [2]). So, in a pure Alloy idiom, we can 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` as a sub-set of `Feature`). Lastly, a fact restricts which variants are considered valid, in the case of the 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 KeyOnPosition and
KeyState.state not in KeyOnPosition 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 lead 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 condition, but since `DarknessModeSwitchOn` is empty in variants without feature `ArmoredVehicle`, the test would be trivially true and permanently disable ambient lighting.

For an example regarding the USA and Canada market variants, during direction blinking 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 Alloy 6 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 [20] 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 [11]. Annotative approaches are better suited for small granularity variability points, which in our experience in modelling systems in Alloy is often the case at the Alloy level of abstraction. Such is the case in the examples above where one needs to change part of a formula or expression, rather than replace the predicate altogether, as would be the case in compositional approaches. In this section we attempt to model the multiple variants of ELS with this lightweight annotative approach. For more details and the formal semantics, the reader is redirected to [20].

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 🛈 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 ①*p*① **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, since they occur in a conjunction (**and**). During analysis, annotated expressions are expanded into plain Alloy by the colorful Analyzer to a style similar to the one presented above, introducing also signatures to model the feature model under the hood. As a consequence, performance of the analyses is expected to be similar in these two approaches.

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
  ①②false②① and ②③false③②
  ①③false③① and ❶~~❷❸false❸❷~~❶ }
```

where, for instance, formula ①②`false`②① forbids the coexistence of USA and Canada market codes, and ❶❷❸`false`❸❷❶ forces the selection of at least one market code[4]. At the level of abstraction of Alloy, 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 Alloy 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 feature 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:

```
LowBeamLeft.state’ in
  ③On③+
  ❸~~BlinkLeft.state’ not in OffP implies~~
  ~~ Half else On~~❸ and
LowBeamRight.state’ in
  ③On③+
  ❸~~BlinkRight.state’ not in OffP implies~~
  ~~ Half else On~~❸
```

where the low 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 branch of a union expression is interpreted as the empty relation when the presence conditions do not hold.

[4] Alloy does not natively support Boolean expressions by design choice, so here `false` is a user-defined predicate with a trivially unsatisfiable formula, such as **`some none`**.

4 Validation & Verification

The Alloy 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.

4.1 Animation and Validation

Validation scenarios Animation commands allow 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 [8]. 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 steps
```

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 `steps`). Since this is a simple scenario, 5 states are sufficient to represent it. 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 'Next Config' to change the configuration of the trace (here, the selected variant), 'Next Init' to change initial state if the trace, or 'Fork' to change the currently focused transition [6].

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 pure Alloy variability idiom the part of this environment predicate could be specified as:

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

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 KeyOnPosition;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 `Alloy` this is defined through the command scope as:

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

Theme customizations The proper graphical representation of instances is key to promote the interpretation of the model among interested parties. The Alloy 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 past 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

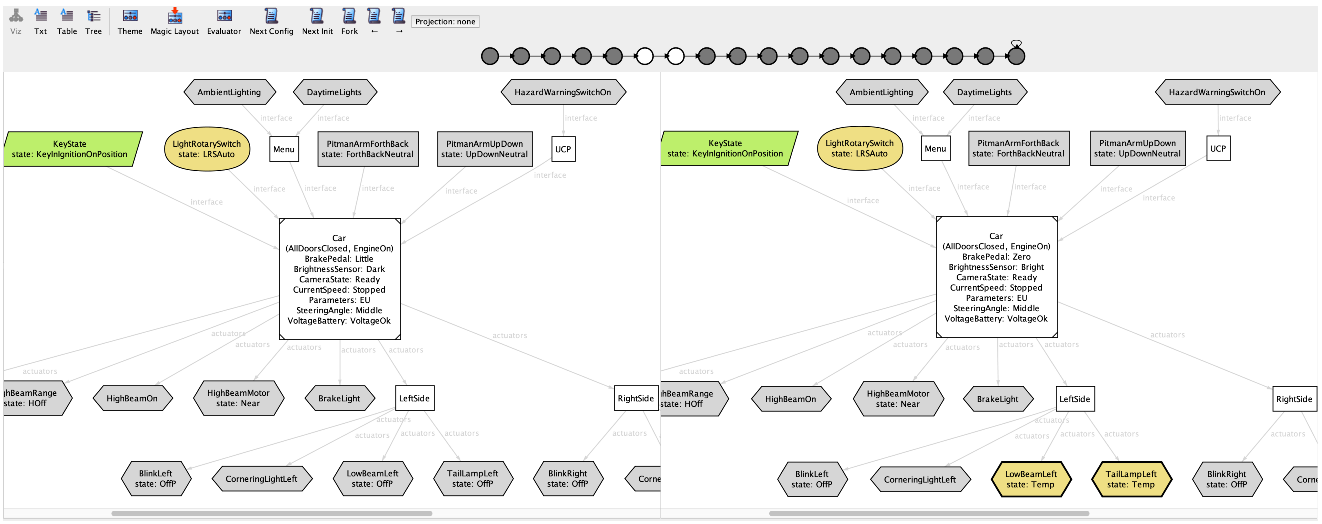

Fig. 2: A step of sequence 1 in the Analyzer under the developed theme.

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 not structured, we introduce additional elements to 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` (white boxes in Fig. 2):

```
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+...) +
  RightSide → (BlinkRight+LowBeamRight+...) }
```

Auxiliary sets (unary relations) 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 elements in Fig. 2):

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

The theme file is available alongside the model specification in the repository.

4.2 Reference validation sequences

To effectively validate the developed model we checked its behaviour against that of the reference validation sequences [14]. 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 Alloy 6 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 3.2, that all signals are bottom-level signatures of the hierarchy on `Signal` with the exact same name as that of the sequence header, and that elements representing the discretized values are at the second layer of the `State` signature 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 Alloy variability idiom, forcing the exact value of signature `Variant`; adapting it to the colorful extension would be straightforward, by specifying the desired variant in the scope of the command.

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 **`steps`** 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 of the document 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.

4.3 Requirement verification

The last step of the process was to verify whether the expected requirements hold for the modelled ELS.

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 KeyOnPosition 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 { always (
  let keyPos = KeyState.state in KeyOnPosition |
  (DaytimeLights in SignalOn and
  not AmbientLighting in SignalOn and
  keyPos) implies
    (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, either the engine is eventually turned off and the low beams are active until then (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 Alloy 6 under the Glucose SAT solver and for 15 **`steps`** 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.

5 Discussion and lessons learned

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 of the reference document has been released after the other 10 issues were reported (unofficially, at least 3 resulted in validation sequence fixes). Roughly, the issues encountered regarded one of the following components:

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;

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	...
...	...	...	...	...	...	...	...	...	...	...

```
let s1 = not AmbientLighting in SignalOn |
always s1
let s1 = not DarknessModeSwitchOn in SignalOn |
always s1
let s1 = LightRotarySwitch.state in LSAuto, s0 = LightRotarySwitch.state in LSOn,
    s2 = LightRotarySwitch.state in LSOff |
s2;s2;s2;s1;s1;s1;s1;s1;s1;s1;s0;s0;s1;s0;s0;s0;always s1
let s0 = BrightnessSensor.state in Dark, s1 = BrightnessSensor.state in Grey,
    s2 = BrightnessSensor.state in Bright |
s2;s2;s2;s2;s1;s0;s2;s0;always s2
EU in Variant
ArmoredVehicle not in Variant
...
after {
  let s2 = LowBeamLeft.state in LightLow, s3 = LowBeamLeft.state in LightOff,
      s1 = LowBeamLeft.state in LightHalf, s0 = LowBeamLeft.state in LightFull |
  s3;s3;s3;s3;s3;s0;s0;s0;s3;s3;s0;s3;s3;s1;s3;s2;always s3
  ... }
```

Fig. 3: Alloy 6 encoding of the sequence from Table 1.

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 assigned 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 perspective, 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

Due to being based on input and output signals, the ELS does not require rich data structures, one of the main advantages of Alloy's relational formalism. Nonetheless, we argue that the Alloy type system still allowed us to handle certain characteristics of the ELS more elegantly than in other lower-level languages. The signature hierarchy allowed us to group together related elements use first-order logic constraints to reason about them. It also allowed us to easily handle changes in the architecture entailed by the different ELS variants. On the other hand, the flexibility of the Alloy language allowed us to encode an underspecified ELS state machine, complemented with additional temporal restrictions that further refine its behaviour.

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 numeric values are not considered. 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. This abstraction also made requirements related to periodic events – such as the bright and dark cycles of blinking lights – the most cumbersome to specify. 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.

The multiple variants of the ELS requirements motivated the extension of the feature annotations from [20] for Alloy 6 and its Analyzer. Since the ELS is not particularly rich in variability, we did not find multi-variant modelling in a pure Alloy idiom to be unmanageable, but it did affect the comprehension of the model. Nonetheless, in the colourful Alloy model variability points are explicitly marked and are thus more maintainable. 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 Alloy – namely, if-then-else expressions common in the definition of state machines, when certain branches are only relevant in certain variants.

A limitation of Alloy-based approaches is the lack of support for deriving a software implementation from the design models. This is particularly challenging in Alloy because the evolution of the system model is constrained by arbitrary temporal constraints and not through events that update the current state. It would be interesting to explore whether under stricter event idioms this would be feasible, such as the one proposed in [4]. However, since the Alloy Analyzer is able to exhaustively generate instances conforming to imposed restrictions, it is well-suited to generate test cases, and in particular has often been used in the *model-based testing* of system implementations.

6 Comparison

Arcaini et al. [3] proposed modelling the ELS (and the SCS) on ASM supported by the ASMETA toolset. The notion of ASM module is used to keep the design modular, and a refinement-based approach is followed to incrementally introduce new functionalities. Like our approach, time is abstracted, with a function notifying that a certain amount of time relevant of an event has passed. For validation they relied on an interactive animator AsmetaA and on the validation of scenarios (written on a specialized language) using AsmetaV. They also developed a technique to export animation sessions into scenarios. The Alloy 6 Analyzer also allows an instance to be exported back as predicate that represents it. Other than ours, this answer to the case study is the only one that has explicitly addressed the variability of the ELS, namely by introducing additional variables in the initial state. The authors argue that this hinders readability and maintainability, and question whether a parametric version of ASM could be helpful. While ASMETA has support for deriving implementations or checking the conformance of available implementations, such features were not explored in this work.

Mammar et al. [24] propose to model the ELS using Event-B supported by the Rodin platform. As typical in the B method, stepwise refinement was employed, although the authors argue that due to the nature of the ELS, it was difficult to identify a manageable refinement strategy. The ProB plugin was used to animate the validation sequences and to model check the developed model, and the Atelier-B plugin to subsequently prove correctness of the specification (with 23% proof obligations automatically proved). Time is modelled explicitly, with an event incrementing time arbitrarily (and in refined versions, triggering timed events). Event-B does not support temporal logic and although ProB allows temporal properties to be verified over Event-B specifications, its performance was infeasible for the ELS. Thus, ProB was only used to check invariants (some temporal properties were addressed by introducing additional variables in the model, but it is not clear which certain requirements were left unchecked due to this limitation). A similar strategy was employed by Mammar and Frappier [23] to model the SCS sub-system of the case study.

Leuschel et al. [19] also rely on the B method to model the blinking lights function of the ELS, but take a different approach. Due to the restrictions on modelling events in Event-B (and the fact that Event-B models are not pure text hindering collaborative work), they start instead by modelling and validating the system in the more flexible classical B in Atelier-B. Time is modelled explicitly as a integer variable that can increase arbitrarily until deadlines for other events are met. During that first step, they relied on ProB for animation and model checking of invariants. One of the main features of ProB is the ability to plugin different visualizers, and the authors developed a new one based on SVGs, VisB, that allowed the visualization of scenarios using the images from the reference document [13]. It is exemplified with one of the validation sequences. Some efforts have been developed to provide more advanced visualizations for Alloy instances [12,7,10], although we have not explored them in this work. Once the classical B model was stable, it was converted into Event-B so that the proof abilities of Rodin could be used (89% automatically proved). These were complemented with model checking of LTL temporal properties, which in ProB are written in a specific syntax, although it's not clear which ELS requirements are covered.

On a completely different approach, Krings et al. [18] propose instead to start from a low-level implementation of the ELS and SCS in MISRA C and subsequently verify its correctness using the CBMC model checker. Test-driven development is employed, including the codification of the validation sequences as integration tests. CBMC does not support temporal logic, but rather relies on code assertions amounting to invariant checking (although additional variables can be used to compare past states). To allow the verification of timed requirements, a mockup clock was used to allow arbitrary evolution of reported time.

7 Conclusions

In this paper we proposed the modelling of the ELS on Alloy 6 and its validation and verification by the accompanying Analyzer. We believe that the flexibility of the Alloy language, allowing a modelling style that mixes under-specified transitions with declarative temporal constraints, is well-suited for early development stages where the concrete behaviour of the system is still not fully specified. This flexibility also allows structural and behavioural variability points to be address even in pure Alloy 6, although with some toll on readability. The early validation of such designs is supported by the Analyzer, which is able to quickly animate scenarios encoded by the user and then perform interactive scenario exploration. This is particularly useful for communicating with the stakeholders less familiar with formal software development methods, despite the fact that the Analyzer's visualizer is not as customizable as, for instance, that of ProB. Verification can then be performed through model checking for arbitrary LTL properties. When compared to the other answers to the case study, Alloy seems to provide a more unified experience: the system model, the scenarios, the temporal properties and the animation and checking commands, are all specified in the same format and processed by the Analyzer.

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. Apel, S., Speidel, H., Wendler, P., von Rhein, A., Beyer, D.: Detection of feature interactions using feature-aware verification. In: ASE, pp. 372–375. IEEE Computer Society (2011)
3. Arcaini, P., Bonfanti, S., Gargantini, A., Riccobene, E., Scandurra, P.: Modelling an automotive software-intensive system with adaptive features using ASMETA. In: ABZ, *LNCS*, vol. 12071, pp. 302–317. Springer (2020)
4. Brunel, J., Chemouil, D., Cunha, A., Hujsa, T., Macedo, N., Tawa, J.: Proposition of an action layer for Electrum. In: ABZ, *LNCS*, vol. 10817, pp. 397–402. Springer (2018)
5. 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)
6. 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)
7. Couto, R., Campos, J.C., Macedo, N., Cunha, A.: Improving the visualization of Alloy instances. In: F-IDE@FLoC, *EPTCS*, vol. 284, pp. 37–52 (2018)
8. Cunha, A., Macedo, N.: Validating the Hybrid ERTMS/ETCS Level 3 concept with Electrum. International Journal on Software Tools for Technology Transfer (2019). DOI 10.1007/s10009-019-00540-4
9. Cunha, A., Macedo, N., Liu, C.: Validating multiple variants of an automotive light system with electrum. In: ABZ, *LNCS*, vol. 12071, pp. 318–334. Springer (2020)
10. Dyer, T., Jr., J.W.B.: Sterling: A web-based visualizer for relational modeling languages. In: ABZ, *LNCS*, vol. 12709, pp. 99–104. Springer (2021)
11. 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)
12. Gammaitoni, L., Kelsen, P.: Domain-specific visualization of Alloy instances. In: ABZ, *LNCS*, vol. 8477, pp. 324–327. Springer (2014)
13. Houdek, F., Raschke, A.: Adaptive exterior light and speed control system (2019). V1.17
14. Houdek, F., Raschke, A.: Validation sequences for ABZ case study "adaptive exterior light and speed control system" (2019). V1.8
15. Houdek, F., Raschke, A.: Adaptive exterior light and speed control system. In: ABZ, *LNCS*, vol. 12071, pp. 281–301. Springer (2020)
16. Jackson, D.: Software Abstractions: Logic, Language, and Analysis, revised edn. MIT Press (2012)
17. Kästner, C., Apel, S., Kuhlemann, M.: Granularity in software product lines. In: ICSE, pp. 311–320. ACM (2008)
18. Krings, S., Körner, P., Dunkelau, J., Rutenkolk, C.: A verified low-level implementation of the adaptive exterior light and speed control system. In: ABZ, *LNCS*, vol. 12071, pp. 382–397. Springer (2020)
19. Leuschel, M., Mutz, M., Werth, M.: Modelling and validating an automotive system in classical B and event-b. In: ABZ, *LNCS*, vol. 12071, pp. 335–350. Springer (2020)

20. 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)
21. Macedo, N., Brunel, J., Chemouil, D., Cunha, A.: Pardinus: A temporal relational model finder. J. Autom. Reason. **66**(4), 861–904 (2022)
22. 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)
23. Mammar, A., Frappier, M.: Modeling of a speed control system using event-b. In: ABZ, *LNCS*, vol. 12071, pp. 367–381. Springer (2020)
24. Mammar, A., Frappier, M., Laleau, R.: An event-b model of an automotive adaptive exterior light system. In: ABZ, *LNCS*, vol. 12071, pp. 351–366. Springer (2020)
25. Post, H., Sinz, C.: Configuration lifting: Verification meets software configuration. In: ASE, pp. 347–350. IEEE Computer Society (2008)