Technical Report:
Syntactic Checks for Safety Cases

Software Engineering Group, Department of Computer Science,
University of Toronto, Toronto, Canada

November 2019

1. Introduction

This technical report presents, by means of example, how syntactic checks on model instances can be formulated using the Object Constraint Language (OCL) and executed using the Eclipse IDE. In particular, this technical report focuses on syntactic checks for assurance cases (or equivalently, safety cases) in the automotive domain, thus the checks are derived not only from assurance case standards (namely, the GSN standard) but also from ISO 26262.

The remainder of this technical report is structured as follows. Since OCL constraints relate to specific metamodels, the adopted safety case metamodel will be presented in Section 2 together with a short introduction to OCL. This is followed by the list of implemented checks in

Section 3, and in Section 4, its implementation and execution in Eclipse will be shown using screenshots. Finally, the technical report concludes with a discussion on other possible types of constraint checks.

2. Background

2.1 Safety Case Metamodel

Figure 1 shows the adopted safety case metamodel, which is based on the GSN standard. Thus as expected, a safety case is modelled to contain goals, strategies, solutions, contexts, justifications and assumptions, all of which are connected to each other by either supported-by relations or in-context-of relations. However, it is extended for the automotive domain to include ASILs as well as independence goals and ASIL decomposition strategies. To support impact assessment, safety case nodes also contain validity attributes and can be associated with impact annotations.

2.2 Object Constraint Language (OCL)

OCL is a formal language for specifying expressions on models. These expressions do not have side effects and are generally used for querying models or specifying constraints over them, the latter of which is the purpose of using OCL in this technical report. Typically, an OCL constraint is as follows (ignoring comments which start with -- and continue to the end of the line):

```
context Goal inv:
      self.supportedBy ->
            forAll(s |
                  s.premise.oclIsKindOf(Goal) or
                  s.premise.oclIsKindOf(Strategy) or
                  s.premise.oclIsKindOf(Solution));
```

`context Goal inv:` specifies that the accompanying OCL expression is an invariant that applies to all instances of `Goal`, thus `self` in the subsequent line refers to the goal model element being checked. `self.supportedBy` traverses the `supportedBy` relation in goals, which in this case results in a set of `SupportedBy` model elements (see Figure 1). The collection operator `->` and `forAll` iterator causes an iteration to be performed on the `SupportedBy` model elements, returning true if they all satisfy the condition specified inside the `forAll` iterator. In this case, the condition is that the premise of the `SupportedBy` element conforms to type `Goal`, `Strategy` or `Solution`.

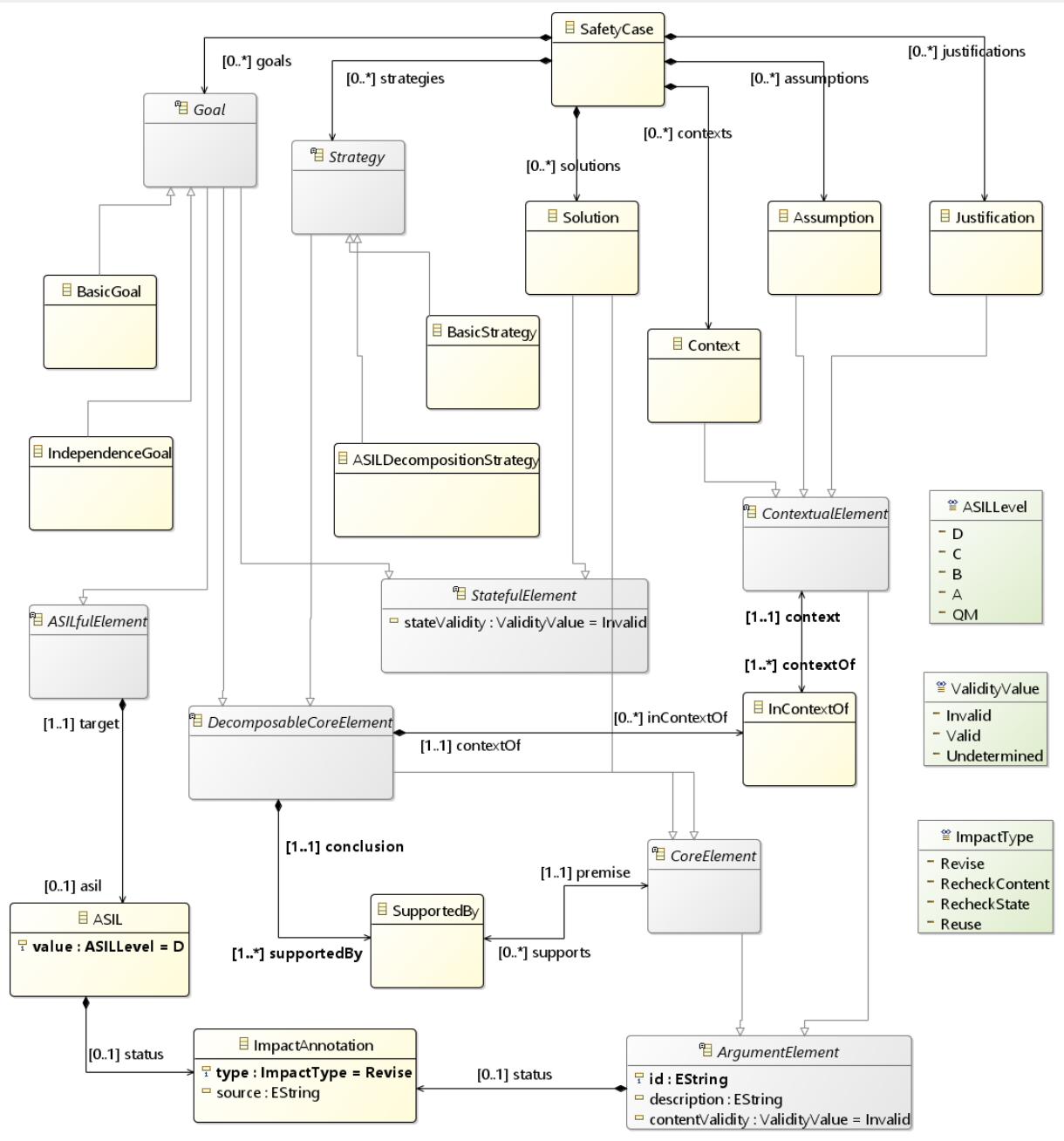


Figure 1. The adopted safety case metamodel

3. Syntactic Checks

In total, 17 syntactic checks have been identified and formalized in OCL. 12 are derived from the GSN standard and therefore relate to safety cases in general, while 4 are derived from ISO 26262 and relate to ASIL decomposition and inheritance. The last syntactic check relates to the validity states of goals and solutions and is specific to the adopted metamodel.

For ease of presentation, this section is divided into five different sub-sections, with each check documented as follows:

Check No. **Check name** (Source of check, e.g. GSN standard, ISO 26262, etc.).
(E.g. 13) Description of syntactic check (in natural language).
`Implementation of check in OCL`

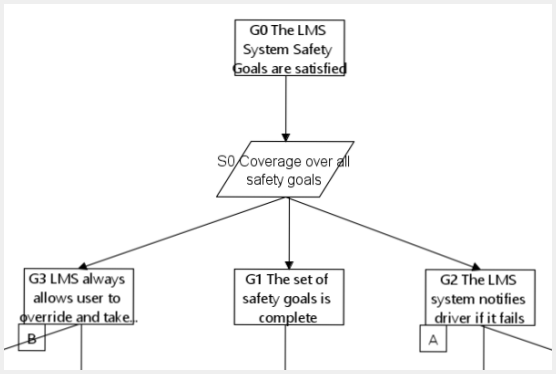
`Pass Case`	`Fail Case`
Fragment of the safety case for the lane management system (LMS) that illustrates the constraint. If no such fragment exists, then a contrived example will be used instead. The complete safety case can be found in Appendix A.	Example safety case that fails the syntactic check. If the check is enforced automatically by the implemented metamodel, then no such model can be instantiated. Model elements that violate the check will be indicated by a cross.

3.1 Supported-By Relations

1. **Goal Supporter** (GSN Standard)
   Goals can only be supported by goals, strategies and solutions.

```
context Goal inv:
      -- Retrieve the supported-by relations of the goal
      self.supportedBy ->

      -- Check that the premise of each supported-by relation
      -- is either a goal, strategy or solution.
            forAll(s | s.premise.oclIsKindOf(Goal) or
                  s.premise.oclIsKindOf(Strategy) or
                  s.premise.oclIsKindOf(Solution));
```

Pass Case



Fail Case

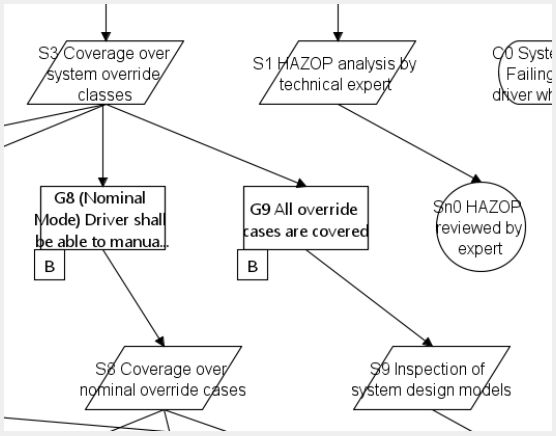
N/A

2. **Strategy Supporter** (GSN Standard)

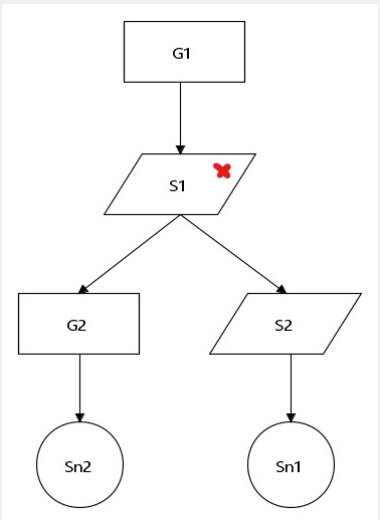
Strategies can only be supported by goals and solutions.

```
context Strategy inv:
      -- Retrieve the supported-by relations of the strategy
      self.supportedBy ->

      -- Check that the premise of each supported-by relation
      -- is either a goal or solution (i.e. not a strategy).
         forAll(s |
               s.premise.oclIsKindOf(Goal) or
               s.premise.oclIsKindOf(Solution));
```

Pass Case



Fail Case

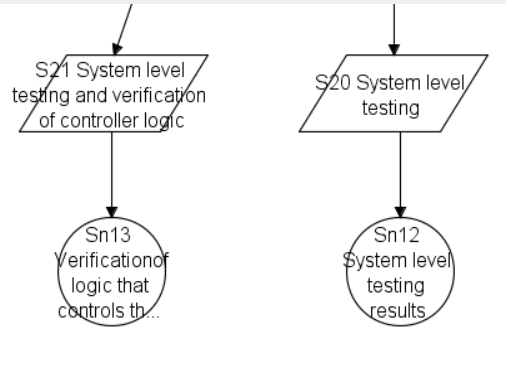


3. **Solution Supporter** (GSN Standard)

Solutions cannot be supported by any safety case element. (Solutions must be leaves)

```
context Solution inv:
      -- Check that the solution is not (more specifically,
      -- cannot be cast into) a decomposable core element and can
      -- therefore not be supported.
      -- This is trivially true.
      self.oclAsType(DecomposableCoreElement).oclIsInvalid();
```

Pass Case



Fail Case

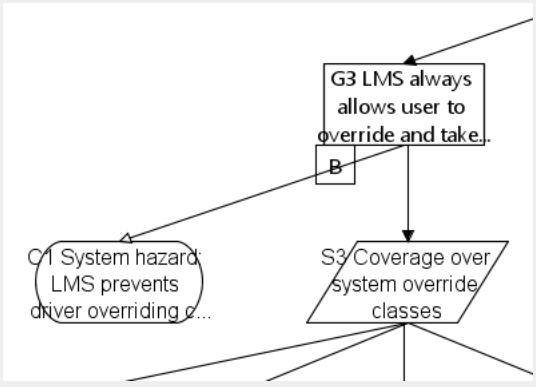
N/A

4. **Contextual Element Supporter** (GSN Standard)

Contextual elements cannot be supported by any safety case element.

```
context ContextualElement inv:
      -- Check that the contextual element is not (more
      -- specifically, cannot be cast into) a decomposable
      -- core element and can therefore not be supported.
      -- This is trivially true.
      self.oclAsType(DecomposableCoreElement).oclIsInvalid();
```

Pass Case



Fail Case

N/A

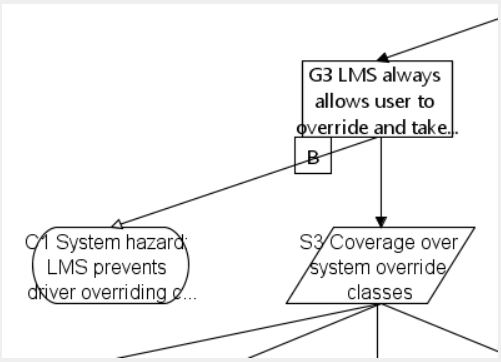
3.2 In-Context-Of Relations

5. **Goal Context** (GSN Standard)

Goals can be in the context of contexts, assumptions and/or justifications.

```
context Goal inv:
      -- Retrieve the contextual elements connected to the goal.
      self.inContextOf.context ->

      -- Check that each contextual element is either a context,
      -- assumption or justification.
            forAll(c |
                  c.oclIsKindOf(Context) or
                  c.oclIsKindOf(Assumption) or
                  c.oclIsKindOf(Justification));
```

Pass Case



Fail Case

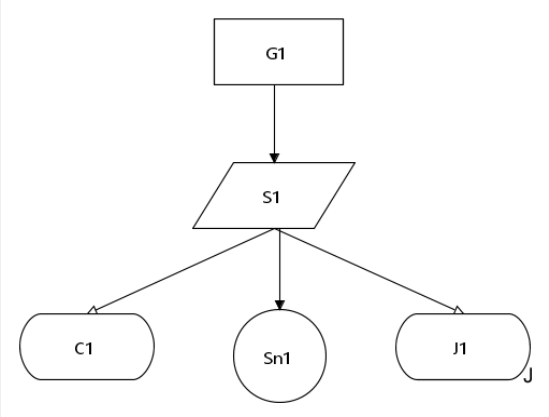
N/A

6. **Strategy Context** (GSN Standard)

Strategies can be in the context of contexts, assumptions and/or justifications.

```
context Strategy inv:
      -- Retrieve the contextual elements of the strategy.
      self.inContextOf.context ->

      -- Check that each contextual element is either a context,
      -- assumption or justification.
            forAll(c |
                  c.oclIsKindOf(Context) or
                  c.oclIsKindOf(Assumption) or
                  c.oclIsKindOf(Justification));
```

Pass Case



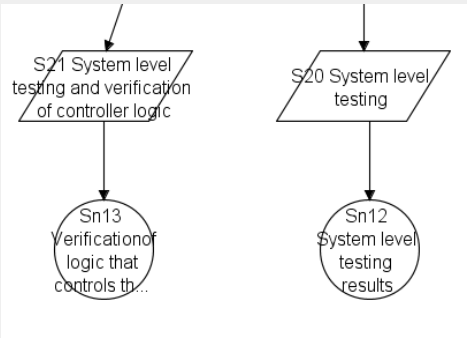
Fail Case
N/A

7. **Solution Context** (GSN Standard)

Solutions cannot be in the context of any safety case element.

```
context Solution inv:
      -- Check that the solution is not a decomposable core
      -- element and can therefore not be contextualised.
      -- This is trivially true.
      self.oclAsType(DecomposableCoreElement).oclIsInvalid();
```

Pass Case



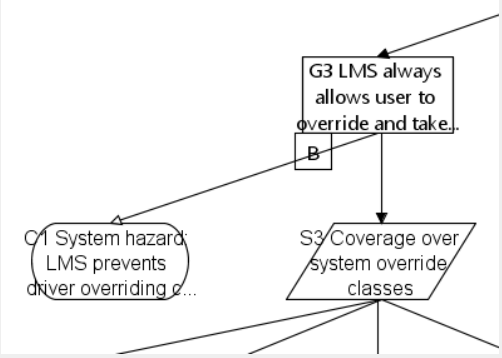
Fail Case
N/A

8. **Contextual Element Context** (GSN Standard)

Contextual elements cannot be in the context of any safety case element.

```
context ContextualElement inv:
      -- Check that the contextual element is not a decomposable
      -- core element and can therefore not be contextualised.
      -- This is trivially true.
      self.oclAsType(DecomposableCoreElement).oclIsInvalid();
```

Pass Case



Fail Case

N/A

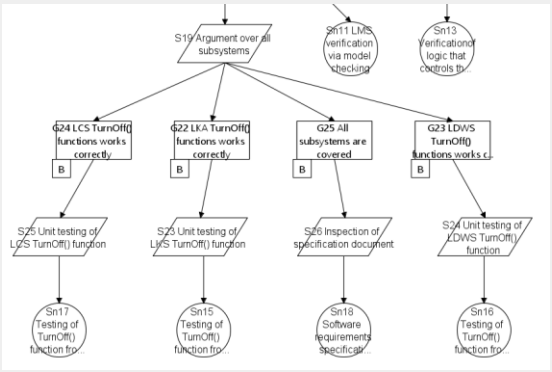
3.3 Overall Structure

9. **Support Cycle** (Implied by GSN Standard)

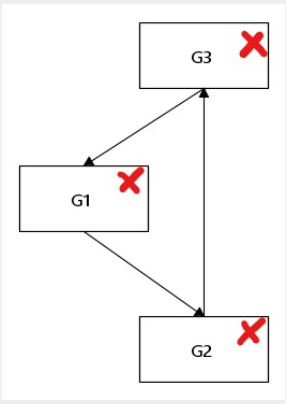
There cannot be any supported-by cycles.

```
context DecomposableCoreElement inv:
      -- Retrieve all supporting elements.
      self.supportedBy.premise ->

      -- Retrieve all their descendants (including themselves).
         closure(p |
              if p.oclIsKindOf(DecomposableCoreElement) then
                   p.oclAsType(DecomposableCoreElement).
                   supportedBy.
                   premise
               else p.oclAsSet()
               endif

      -- Check that none of the retrieved supporting elements is
      -- the decomposable core element itself.
         ) -> excludes(self);
```

Pass Case



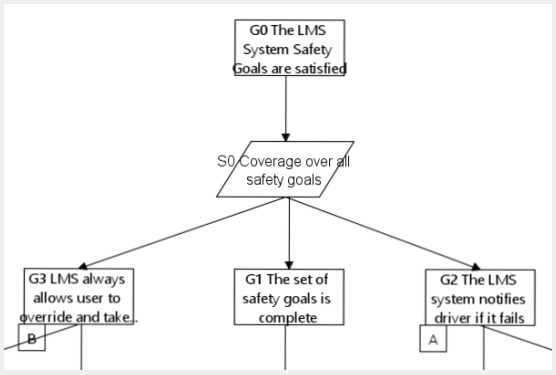
Fail Case



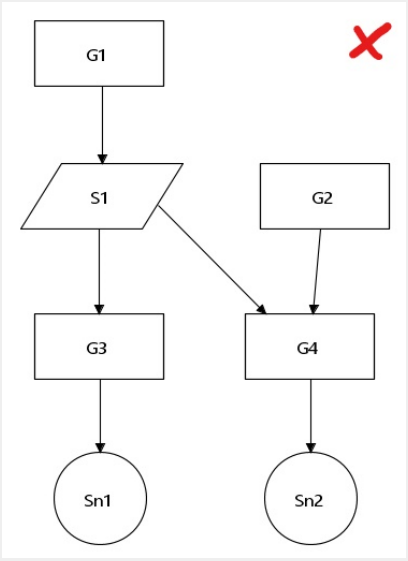
10. **Single Root** (Implied by GSN Standard)
    There can only be one root in an assurance case.

```
context SafetyCase inv:
      -- Retrieve all core elements that are not supporting
      -- any other (decomposable) core element.
      CoreElement.allInstances() ->
            select(d |
                  d.supports.conclusion -> isEmpty()

      -- Check that there is only one such core element.
            )-> size() = 1;
```

Pass Case



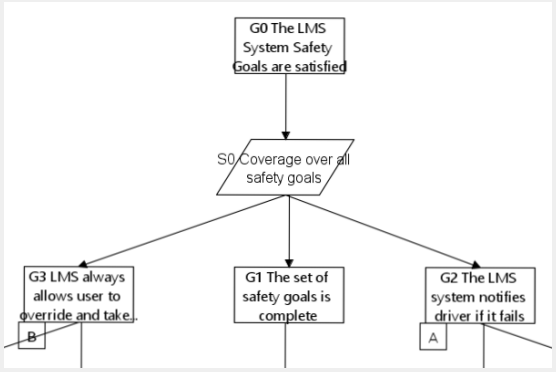
Fail Case



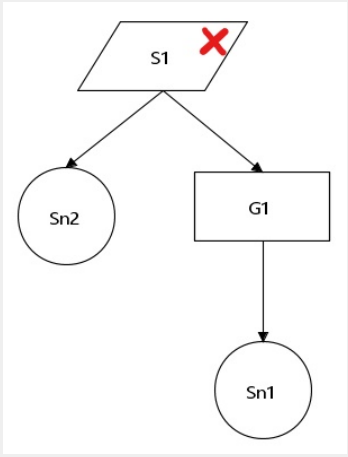
11. **Goal Root** (Implied by GSN Standard)
    The root of an assurance case must be a “basic” goal.

```
context CoreElement inv:
      -- Retrieve the element supported by the core element.
      self.supports.conclusion ->

      -- If the core element is not supporting anything,
      -- it is the root and must therefore be the goal.
            isEmpty() implies self.oclIsTypeOf(BasicGoal);
```

Pass Case



Fail Case

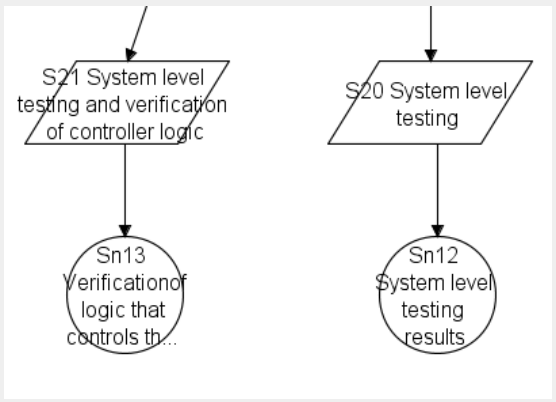


12. **Non-Decomposable Leaves** (Implied by GSN Standard)

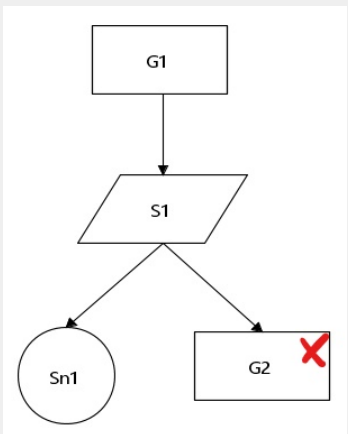
    The leaves of an assurance case must be solutions, not goals nor strategies.

```
Context DecomposableCoreElement inv:
      -- The decomposable core element (i.e. goal or strategy)
      -- must be supported by a (non-null) safety case element.
      self.supportedBy.premise -> size() > 0 and
            self.supportedBy.premise -> excludes(null);
```

Pass Case



Fail Case

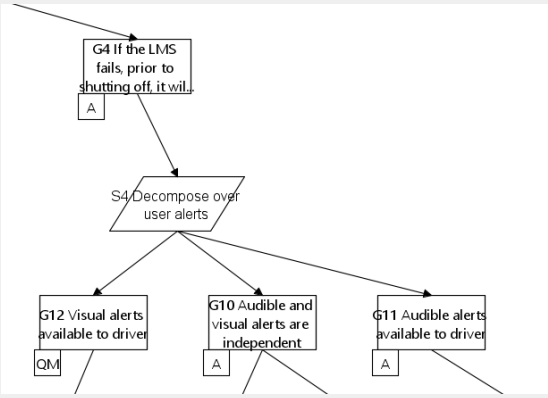


3.4 ASIL Decomposition

13. **ASIL Decomposition Independence** (Implied by ISO 26262)
    An ASIL decomposition strategy must be supported by one “independence goal”.

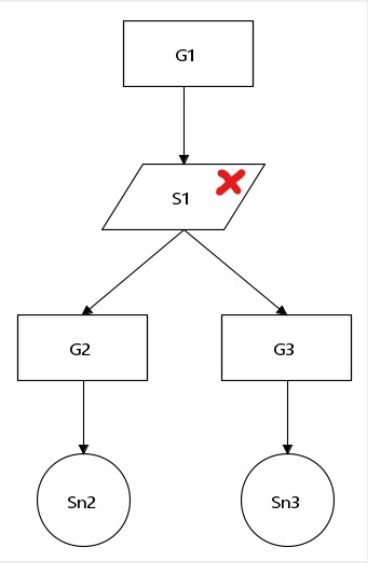
```
context ASILDecompositionStrategy inv:
      -- Retrieve the independence goals supporting the strategy
      self.supportedBy.premise ->
            selectByType(IndependenceGoal) ->

      -- Check that there is exactly one such goal.
            size() = 1;
```

Pass Case



(S4 is the ASIL decomposition strategy and G10 the independence goal)

Fail Case

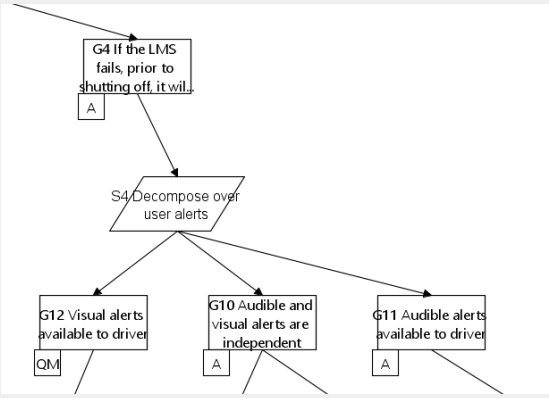


(S1 is the ASIL decomposition strategy while G2 and G3 are both basic goals)

14. **ASIL Decomposition Components** (Implied by ISO 26262)
    An ASIL decomposition strategy must be supported by two “basic” goals

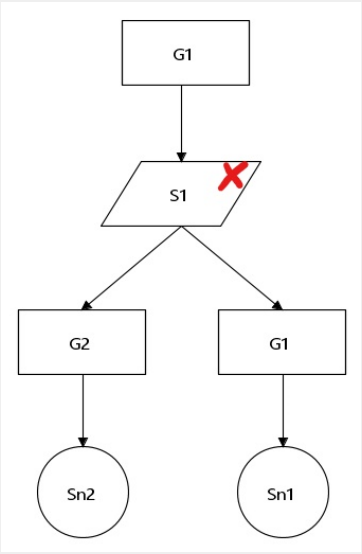
```
context ASILDecompositionStrategy inv:
      -- Retrieve the basic goals supporting the strategy
      self.supportedBy.premise ->
            selectByType(BasicGoal) ->

      -- Check that there are exactly two such goals.
            size() = 2;
```

Pass Case



(S4 is the ASIL decomposition strategy while G11 and G12 are the basic goals)

Fail Case



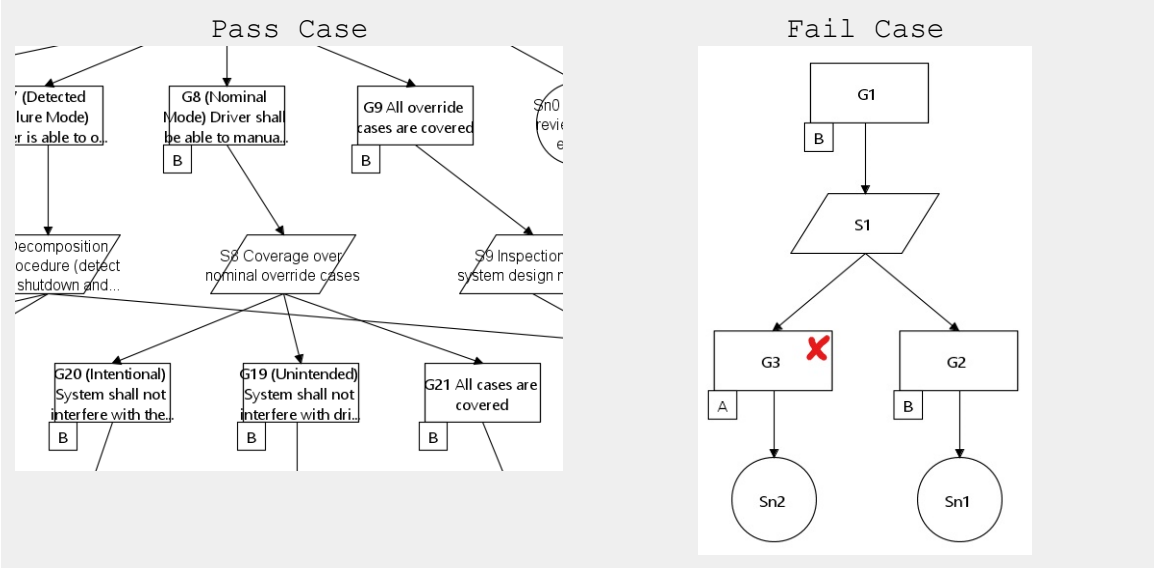
(S1 is the ASIL decomposition strategy and G2 the only basic goal)

15. **ASIL Inheritance** (ISO 26262)

    Any child goal that support a parent goal directly or via a “basic” strategy must have the same ASIL or stronger as the parent goal (if any).

```
context Goal inv:
let
   -- Retrieve all parent goals of the goal in question.
   directParents : Set(Goal) = self.supports.conclusion ->
      select(d |d.oclIsKindOf(Goal)).oclAsType(Goal) -> asSet(),

   -- Retrieve all "indirect" goals by retrieving:
   -- 1) All basic strategies supported by the goal
   -- 2) All goals supported by those strategies.
   indirectParents : Set(Goal) = self.supports.conclusion ->
       select(d |d.oclIsTypeOf(BasicStrategy)).supports.conclusion ->
          select(d |d.oclIsKindOf(Goal)).oclAsType(Goal) ->
             asSet() in indirectParents ->
               union(directParents) ->

   -- Check that each parent's ASIL has been inherited correctly.
   -- 1) If the parent has no ASIL, then it is trivially true.
   -- 2) Otherwise, if the child has no ASIL (but the parent
   --    does), then it is trivially false.
   -- 3) Otherwise, the parent must have a stronger ASIL.
```

```
forAll(g |
    if g.asil = null then true
    else if self.asil = null then false
       else
          g.asil.value = ASILLevel::QM or
         (g.asil.value.toString() <=
             self.asil.value.toString() and
             self.asil.value <> ASILLevel::QM)
       endif
    endif);
```

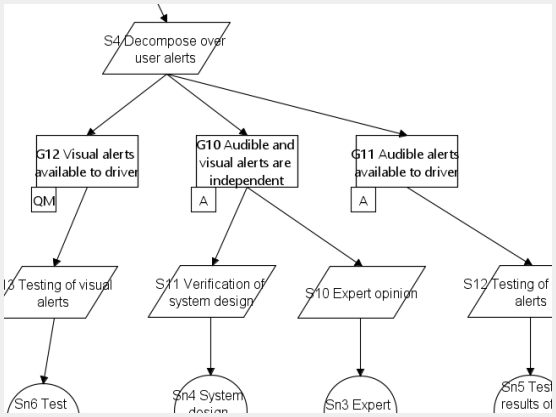


16. **ASIL Descendants** (Implied by ISO 26262)
    The two basic goals supporting an ASIL decomposition strategy cannot share any common descendant goal or solution.

```
context ASILDecompositionStrategy inv:
let
    -- Retrieve the basic goals supporting the strategy.
    -- Returning a sequence (instead of a set) allows the
    -- selection of a specific basic goal to operate on.
    goalSeq: Sequence(CoreElement) = self.supportedBy.premise ->
        select(p | p.oclIsTypeOf(BasicGoal)),

    -- Retrieve all descendants of the first basic goal
    -- (excluding itself).
    g1Descendants : Set(CoreElement) = goalSeq -> at(1) ->
        closure(c |
            if c.oclIsKindOf(DecomposableCoreElement) then
```

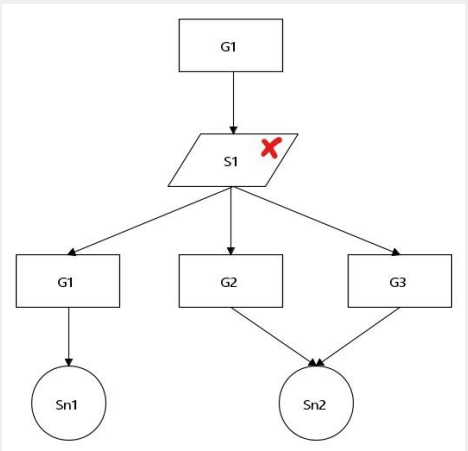
```
            c.oclAsType(DecomposableCoreElement).
                supportedBy.premise
        else
            null
        endif),

-- Retrieve all descendants of the second basic goal
-- (excluding itself).
g2Descendants : Set(CoreElement) = goalSeq -> at(2) ->
    closure(c |
        if c.oclIsKindOf(DecomposableCoreElement) then
            c.oclAsType(DecomposableCoreElement).
                supportedBy.premise
        else
            null

 -- Check that there are no shared descendants.
        endif) in g1Descendants ->
            intersection(g2Descendants) = Set{};
```

Pass Case



(G11 and G12 are the basic goals of the ASIL decomposition strategy S4)

Fail Case



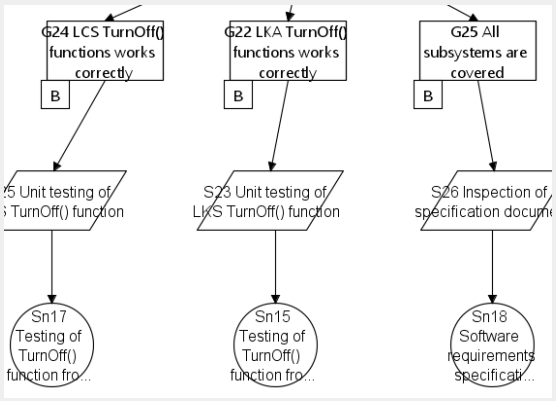
(S1 is the ASIL decomposition strategy while G2 and G3 are the basic goals)

3.5 State Validity

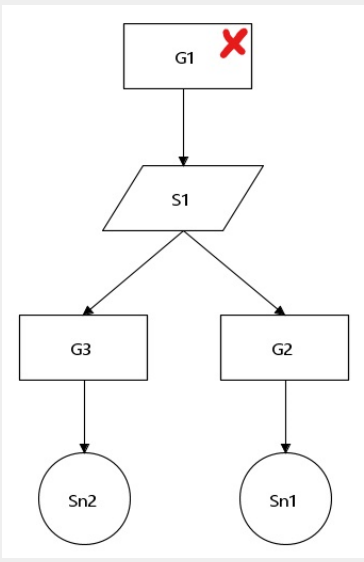
17. **State Validity Inheritance** (University of Toronto)
    If the state of a parent goal is valid, then the states of all child goals and solutions must also be valid (Current metamodel only supports AND decomposition.)

```
context Goal inv:
-- Proceed with check if goal's state is valid.
self.stateValidity = ValidityValue::Valid implies

    -- Retrieve all stateful elements (i.e. goals and solutions)
    -- that are directly supporting the goal in question.
    let directChildren : Set(StatefulElement) =
            self.supportedBy.premise ->
                select(d |d.oclIsKindOf(StatefulElement)).
                    oclAsType(StatefulElement) -> asSet(),

    -- Retrieve all stateful elements that indirectly support the
    -- the goal in question via a strategy.
        indirectChildren : Set(StatefulElement) =
            self.supportedBy.premise ->
                select(d | d.oclIsKindOf(Strategy)).
                    oclAsType(Strategy).supportedBy.premise.
                        oclAsType(StatefulElement) -> asSet() in

    -- Check that the states of all retrieved children are valid.
                            indirectChildren ->
                                union(directChildren) ->
                                    forAll(g |
                                        g.stateValidity =
                                            ValidityValue::Valid);
```

Pass Case



Fail Case



(All states are valid except G2 in the failed case)

4. Demo

The OCL constraints were incorporated into the safety case metamodel by using the OCLinEcore editor in Eclipse, a screenshot of which is shown in Figure 2 below. In particular, this figure illustrates how the four constraints on goals were added to the Goal class in the metamodel, namely Goal Supporter (Check 1), Goal Context (Check 5), ASIL Inheritance (Check 15) and State Validity Inheritance (Check 17).

Figure 3 shows how one can validate a safety case based on the implemented constraints. In this case, the Sirius editor for safety cases is being used to validate the LMS safety case, and this is achieved by right-clicking the diagram and selecting “Validate diagram” from the menu. No constraints were violated by the LMS safety case, but if there were, each violation will be reported as an error, and the corresponding model elements will be flagged with a small red cross. An example is shown in Figure 4, in which strategy S1 violates the constraint Strategy Supporter (Check 2).

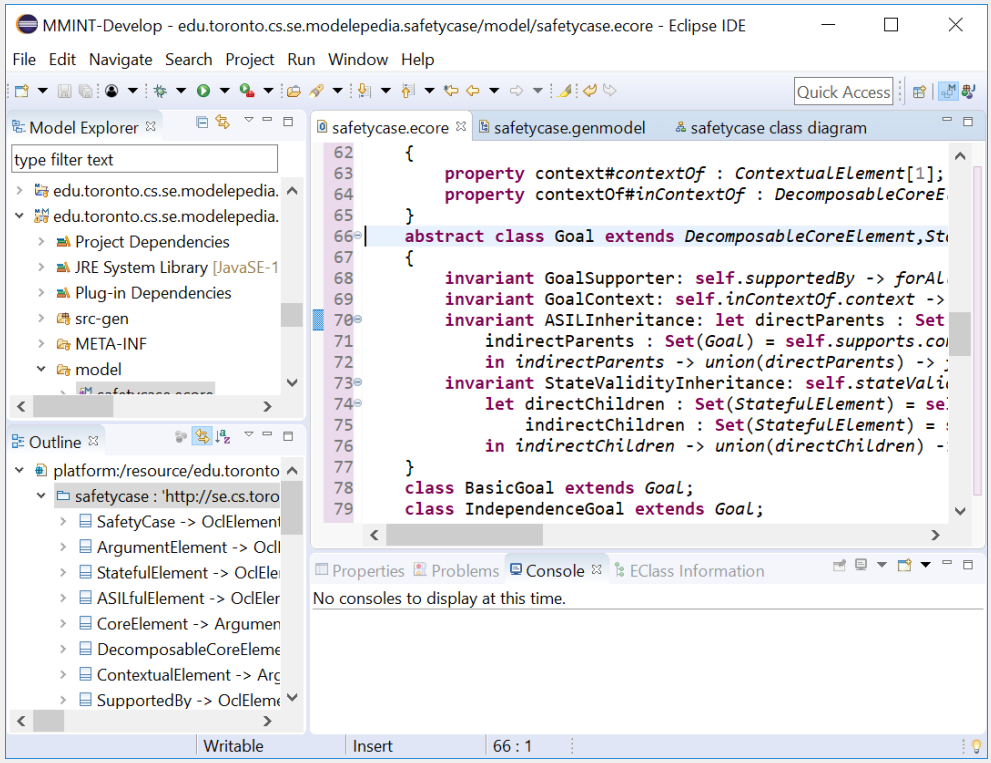


Figure 2. The OCLinEcore Editor in Eclipse

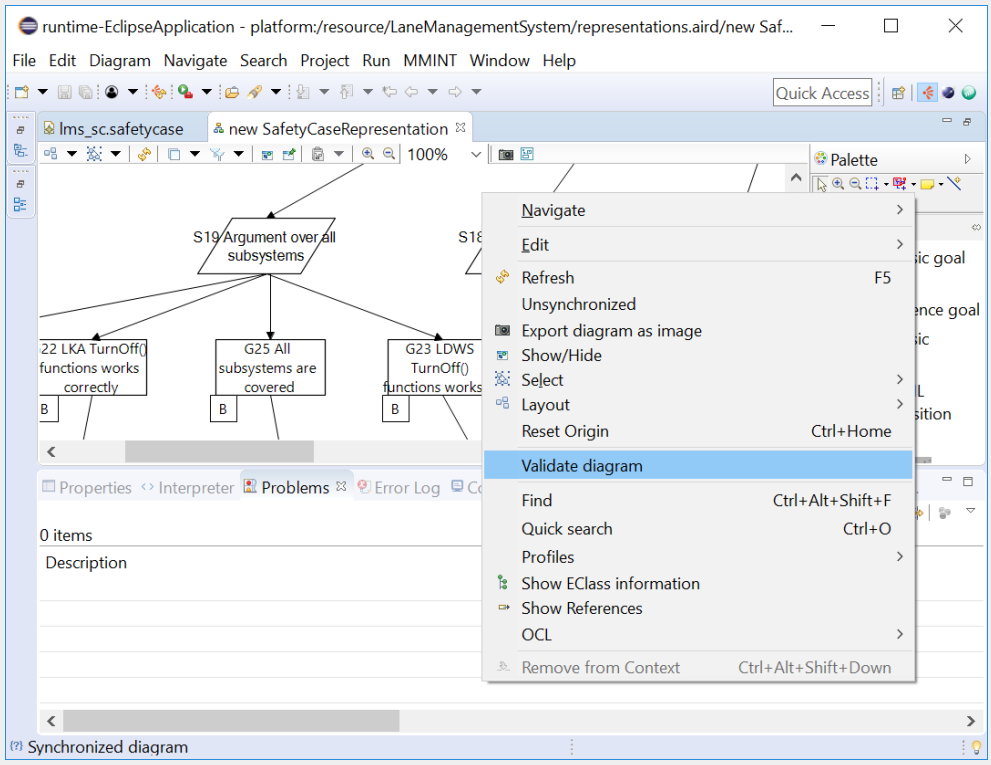


Figure 3. Validation of the LMS Safety Case

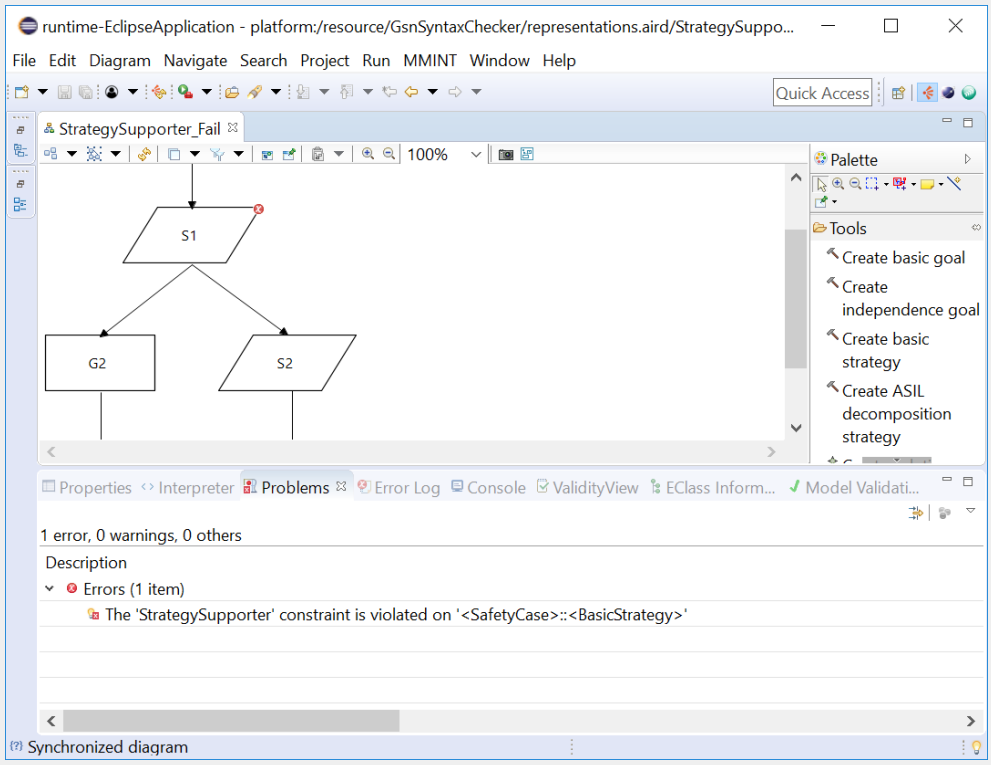


Figure 4. Violation of an OCL Constraint

5. Discussion

The provided list of constraint checks can most certainly be augmented with others. In what follows, we give a list of ideas that have been brought forward but not yet formalized or checked. We expect examples of many of these checks will be implemented as part of the Semantic Checks technical reports.

5.1 Other Possible Safety Case Checks

Here, we consider other safety case checks.

1. ASIL decomposition complies with ISO 26262.
That is, the ASIL of a parent node cannot be higher than the sum of ASIL levels of its children nodes.

2. “Invalid” and “Undetermined” states are inherited correctly.
That is, if a node is invalid or undetermined, its parent should be marked as such as well.

3. Inheritance of state validity that accounts for OR-decomposition.
That is, all the current checks assume AND decomposition. The aim is to repeat them for OR decomposition.

4. A goal should be supported by evidence type appropriate for the corresponding ASIL.
That is, the attached evidence, i.e., test coverage criteria, verification claims, etc., is what is required by the ASIL. The aim is not to check the evidence itself but just the metadata capturing its type.

As well as various process-related checks, the details of which are TBD.

5.2 Categorizing Constraint Checks

We have detailed constraint checks on safety cases in Sec. 3 and have described a more general application of constraint checking in Sec. 5.1. As part of the research, we intend to define a *constraint check taxonomy* to help practioners and tool developers work with them in a systematic manner. There are many potential categories of constraint checks that can be included in a taxonomy and we discuss some here with examples from this document.

All the constraints in Sec. 3 are *intra-model* checks as they are checked on a single model (the safety case). On the other hand, *inter-model* constraints are those which rely on the existence of traceability mappings between models. For example, a check such as “Is there a fault tree analysis (FTA) conducted for all hazards that are ASIL B and above?” While the constraint checking tooling described in this technical report is limited to intra-model checks, options to support inter-model constraint checking are under investigation.

Some constraints deal with the content of artifacts while others with the *process* to create them. For example, the *product* constraints in Sec. 3.1, 3.2, and 3.3 are *well-formedness* checks to ensure that a GSN safety case is meaningful. These are clearly *syntactic* checks. Semantic product constraints can check a broader range of properties, e.g., the consistency between artifacts (e.g., “Are all system functions comprehended in the Hazard Analysis?”). In contrast, a check like “Is there a FTA conducted for all hazards that are ASIL B and above? “is a process constraint that ensures that a particular artifact (in this case, FTA) is produced.

Other kinds of distinctions that are relevant to a taxonomy include: constraints to check correctness vs. completeness, constraints that are necessary vs. sufficient conditions, existential vs. universal constraints, etc. The criteria for including a category in the final taxonomy will be based on how it is relevant to supporting the safety process.

Appendix

A. Safety Case for Lane Management System