Automated Model Based Assurance Case Management Using Constrained Natural Language

Ran Wei, Zhe Jiang*, Haitao Mei*, Konstantinos Barmpis*, Simon Foster, Tim Kelly, and Yan Zhuang

***Abstract*—Assurance cases are used to communicate and assess confidence in critical system properties, e.g, safety and security. Historically, assurance cases have been manually created documents, validated by engineers through lengthy and error-prone processes. Recently, system assurance practitioners have begun adopting model based approaches to improve the efficiency and quality of system assurance activities. This becomes increasingly important, for example, to ensure the safety of Robotics and Autonomous Systems (RAS), as they are adopted into society. Such systems can be highly complex, and so it is a challenge to manage the development life-cycle and improve efficiency, including coordination of validation activities, and change impact analysis in inter-connected system assurance artifacts.**

However, adopting model based approaches requires skills in model management languages, which system assurance practitioners may not be acquainted with. In this paper, we contribute an automated validation framework for model based assurance cases, which promotes the usage of a Constrained Natural Language (CNL), that can be automatically transformed and executed against engineering models involved in assurance case development. We apply our approach to a case study based on an Autonomous Underwater Vehicle (AUV).

***Index Terms*—Safety Critical Systems Engineering, Model Based Assurance Case, Automated Assurance Case Validation.**

I. Introduction

Safety-critical systems require justifications that they are acceptably safe to operate in their defined operational contexts. *Assurance case*s provide an explicit means for arguing, justifying and assessing the confidence in system properties such as safety and security. The submission of an assurance case is increasingly being required during system certification processes in many safety-critical industries, such as aviation [1], nuclear power [2], transportation [3], [4], and defence [5]. Ideally, an assurance case is the central point of reference for all system stakeholders, to allow effective communication and traceability from the assurance case to its referenced engineering artifacts. Prior to certification, an assurance case must be rigorously, and often independently, validated, ensuring the safety arguments and their supporting evidence are coherent and convincing.

Assurance cases are not typically self-contained documents, in the sense that they organise, refer and pull together contextual and evidential information stored in other documents (e.g. requirements, design models, etc.) in order to form an argument about the safety of the systems under question. Hence, the validation of an assurance case involves the validation of the engineering artefacts/documents it depends on/refers to, which is often an informal, manual, and error-prone process [6].

Ran Wei and Zhe Jiang are with University of Cambridge, UK. E-mails: rw741@cam.ac.uk, zj266@cam.ac.uk.

Haitao Mei, Konstantinos Barmpis, Simon Foster, and Tim Kelly are with University of York, UK.

Yan Zhuang is with Dalian University of Technology, China.

* Corresponding authors

Over the past decade, system assurance practitioners have begun adopting *Model-Driven Engineering* (***MDE***). Benefits provided by MDE promise the interoperability, integration and coordination of diverse artifacts/models to provide the basis for an automated, coherent, and self-contained assurance case. However, current assurance case notations, such as the Goal Structuring Notation (GSN) [7] and Claim-Argument-Evidence (CAE) [8], do not have sufficient model based foundations to systematically fulfil such promises. Consequently, existing model based assurance case approaches cannot provide the automated validation of an assurance case and the engineering artifacts that it may depend on. The inspection, validation, and change management of engineering artifacts mostly remain manual. To address this limitation, approaches for maintaining traceability from a model based assurance case to its supporting engineering artifacts have been proposed [9], [10]. However, such traceability links are mere "hyperlinks", in the sense that the validation of the referenced artifacts is still performed manually. To address such problem, we propose an approach and tool support to add validation rules besides the traceability links to refine the traceability to specific parts (i.e. model element(s)) of an engineering artifact. To further promote automation in model based assurance case development, validation rules can be written in model validation languages and embedded to the assurance case to support the automated validation of referenced engineering artifacts within an assurance case.

Whilst automated validation via means of validation rules provides a significant improvement on the efficiency for the validation of the overall assurance case, one more problem arises: system assurance practitioners are often not acquainted with low-level model validation languages used to write validation rules. Stakeholders involved in an assurance case development process typically communicate the validation of engineering artifacts in natural languages (e.g. English). To introduce model validation languages to stakeholders, it typically means that they would spend substantial amount of time to learn such languages before they can perform the validation of assurance cases in an efficient manner. In this paper, we contribute an approach for automated assurance case validation, using which, fine-grained traceability links from an assurance case to its depending engineering model elements can be established with embedded model validation rules. We contribute an approach for expressing validation rules using a Constrained Natural Language (CNL), to promote comprehensibility of the traceability from an assurance case to its supporting engineering model elements.

Our contributions are:

1) An approach and its supporting tool to manage assurance cases and validate their referenced engineering artifacts in an automated manner;

2) A CNL metamodel and grammar that can be used to express validation rules against arbitrary models;
3) Automated means to validate an assurance case and its referenced engineering artifacts by executing validation rules written in CNL;
4) The application of all of above to an AUV case study.

II. Background and Motivation

A. Assurance Cases

The concept of assurance cases has been well-established in the safety-related domains, where the term *safety case* is normally used. For many industries, the development, review and acceptance of a safety case form a key element of regulatory processes. This includes nuclear, defence, aviation and railway industries [11]. Safety cases form the basis for certification in the safety engineering lifecycle, as illustrated in Figure 1. It typically organises all information regarding safety throughout the engineering process, detailed in [6]

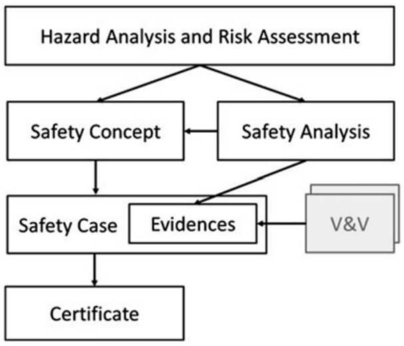


Fig. 1: Safety engineering lifecycle [6].

Historically, safety arguments were communicated in safety cases through free text. However, problems were experienced when text is the only medium available for expressing complex arguments. One problem of using free text is that the language used in the text can be unclear, ambiguous and poorly structured. There is no guarantee that system engineers would produce safety cases with clear and well-structured language. Also, the capability of expressing cross-references with free text is very limited, multiple cross-references can also disrupt the flow of the main argument.

To overcome the problems of expressing safety arguments in free text, graphical argumentation notations were developed. They are capable of explicitly representing the elements that form a safety argument (i.e. requirements, claims, evidence and context), and the relationships between these elements (i.e. how individual requirements are supported by specific claims, how claims are supported by evidence, and the assumed context that is defined for the argument).

One important remark is that an assurance case often refers to other documents in the process of arguing the safety of systems. The references typically serve two purposes:

Contextual: in arguing the safety of systems, practitioners need to refer to other types of documents to provide contextual information. For example, the developer of an assurance case can refer to a Hazard Log document, so that the Hazard Log document can be used for discussion when reviewing the assurance case.

Evidential: the argument regarding the safety of the system typically involves safety goals (or safety requirements) and evidence to support them (i.e. how they are met). Therefore, evidence needs to be referenced in the assurance case to assemble the argument. For example, in order to claim that a component is acceptably safe, the developer of an assurance case can refer to a safety analysis document (e.g. Failure Mode and Effect Analysis - FMEA), which provides proof that the safety-related failures of the component have an acceptably low probability to occur.

Hence, an assurance case is not a self-contained document, in the sense that one cannot validate the assurance case just by looking at the assurance case alone - it refers to, and pull together, information from other documents to form the argument about the safety/security of the system. This becomes particularly problematic in assurance case validations, as practitioners need to trace, navigate to, review and validate the engineering artifacts that assurance case depends on on [12], [13], which is often a time-consuming and error-prone process.

B. Goal Structuring Notation

The Goal Structuring Notation (GSN) [7] is a well-established graphical argumentation notation that is widely adopted within safety-critical industries for the presentation of safety arguments within safety cases. The core elements of GSN are shown in Figure 2.

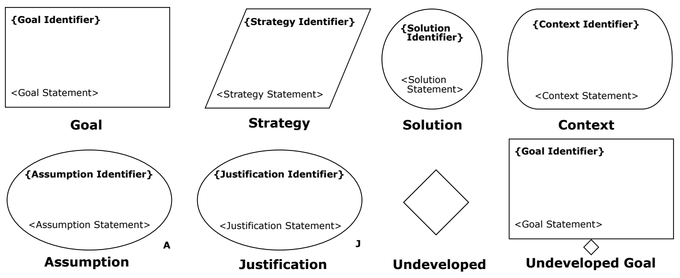


Fig. 2: Core GSN elements.

A *Goal* represents a safety claim within the argumentation. A *Strategy* is used to describe the nature of the inference that exists between a goal and its supporting goal(s). A *Solution* represents a reference to an evidence item or multiple evidence items. A *Context* represents a contextual artefact, which can be a statement or a reference to contextual information. An *Assumption* represents an assumed statement made within the argumentation. A *Justification* represents a statement of rationale. An element can be *Undeveloped*, which means that a line of argument has not been developed yet (meaning it is abstract and needs to be instantiated). The *Undeveloped* notation can apply to *Goal*s and *Strategies*. The *Undeveloped Goal* in Figure 2 is an example. Core elements of GSN are connected with two types of connectors, as shown in Figure 3. The *SupportedBy* connector allows inferential or evidential relationships to be documented. The *InContextOf* relates contextual elements (i.e. *Context*, *Assumption* and *Justification*) to *Goal*s and *Strategies*.

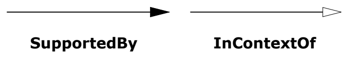


Fig. 3: GSN connectors.

When elements of GSN are linked together in a network, they are often referred to as a *goal structure*. The purpose of a goal structure is to show how *Goal*s are successively broken down into sub-*Goal*s until a point is reached where *Goal*s can be supported by direct reference to available evidence (*Solution*s). An example of a goal structure is shown in Figure 14.

GSN has been adopted in many industries for its power of expressiveness [11] and is being increasingly used in Robotic and Autonomous Systems [14]–[16]. After Uber's 2018 incident, a Safety Case Framework for Aurora's self-driving vehicles has been developed to provide guidance and best practice[1].

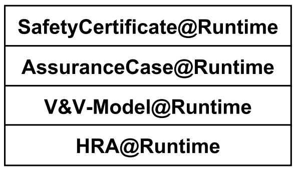


Fig. 4: Models at Runtime on Different Abstraction Levels.

C. Runtime System Assurance

In recent years, Robotic and Autonomous Systems (RAS) emerge. Industries see huge economic potential in such systems - particularly due to their open (RASs connect to each other at runtime) and adaptive (RASs adapt to changing contexts) nature, which enables new types of promising applications in domains such as automotive, healthcare, and home automation [17]. Since the majority of application domains of RASs are safety-critical, it is imperative to assure the safety and/or security of such systems. However, due to the open and adaptive nature of RAS, it becomes impossible to sufficiently anticipate the runtime overall system structure, and the system's operational context at development time [6].

Therefore, existing design time system assurance activities are insufficient to enable dynamic system assurance for RAS at runtime. In [6], [18], the importance of system assurance at runtime for RAS is identified and the idea of *Models@Runtime* on four different abstraction levels are proposed, as shown in Figure 4. As discussed in [19], the ideal balance of system assurance at runtime is on the AssuranceCase@Runtime, in which a system is able to evaluate the safety of itself, as well as other systems that interact with the system at runtime. Thus, there is a need to shift design time assurance case documents to runtime assurance case models to assure open adaptive systems at runtime, which would require the support for automated assurance case validation.

D. Model Based Assurance Cases

Over the past few years, model based assurance case approaches have been proposed due to the benefits introduced by MBSE. Studies have shown how automated MBSE operations can be performed on model based assurance cases (created using GSN) to check the well-formedness of assurance cases [10], generate and assemble structured argumentation within assurance cases [12], and automatically generate texts for assurance case reports [10]. However, existing model based assurance case approaches (GSN and CAE) do not provide sufficient support for traceability to engineering artifacts.

[1]https://safetycaseframework.aurora.tech/gsn

To address the limitations of GSN and CAE, the Object Management Group (OMG) standardised the Structured Assurance Case Metamodel (SACM) [20]. SACM allows the users to define *AssuranceCase* packages, which contain *Argumentation* packages, and additional *Artifact* packages and *Terminology* packages [9]. The additional facilities allow the creation of assurance case models, which contain references to external digital artifacts (more details in SectionIII). This facility is briefly discussed in [9], and a traceability mechanism is discussed. Such traceability mechanism only provides a "hyperlink" that points to the location of the referenced engineering artifact. To provide fine-grained traceability, atop of the "hyperlink", we provide an approach and tool support to store model validation rules in assurance cases created using SACM, and to execute such validation rules to enable automated validation of assurance cases.

Whilst automated model validation is crucial to assurance case development [19], [21], the usage of low-level validation languages can lead to a couple of problems. First, the development and review of an assurance case typically involve stakeholders with different expertise, who may not be acquainted with model validation languages that is used to write the validation rules. This could lead to problems in comprehending and communicating the assurance case, and therefore lead to a decrease in productivity for assurance case development and review. Secondly, fixating on specific validation languages contribute to undesirable technology lock-in for assurance case development and review. For interoperability purposes, not only the metamodel for assurance cases should be defined and standardised (like SACM), it is also necessary to define the language used in assurance cases (presumably for different domains) so that the semantics of the terms/expressions used in the assurance cases are consistent. To address the above problems, in this paper we make a first practical step towards a fully model based assurance case approach by defining and using an executable Constrained Natural Language (CNL) within an assurance case to validate engineering artifacts that the assurance case refers to/depend on.

III. Approach Overview

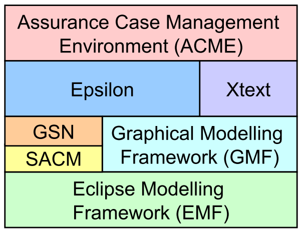

Assurance Case Management Environment (ACME)
Epsilon
Xtext
GSN
SACM
Graphical Modelling Framework (GMF)
Eclipse Modelling Framework (EMF)

Fig. 5: Overview of the Assurance Case Management Environment (ACME).

Our approach is backed by our tool – *Assurance Case Management Environment* (*ACME*), the components of which are illustrated in Figure 5. ACME is an integrated modelling tool which provides graphical editors for creating models conforming to the Structured Assurance Case Metamodel (SACM) [20]. Since SACM is relatively new and has not been widely adopted, we made the design decision to use a GSN metamodel [9] to build the prototype for ACME. In this way, ACME supports the creation of model based

GSN diagrams, and at the same time provides access to all other features of SACM. We implement SACM and GSN with the Eclipse Modelling Framework (EMF) [22], and use **Graphical Modelling Framework (GMF)** [23] to create graphical editors for SACM and GSN. We use **Epsilon** [24] to manage models defined in different modelling technologies (EMF, Simulink, etc). And we use **Xtext** [25] to implement the CNL, together with their generated development tools such as a dedicated editor, content assist and language validation.

It is necessary to discuss SACM concepts relevant to this work to better explain our approach. In SACM, the containing element is *AssuranceCasePackage*, which may contain a number of *TerminologyPackage*s (for terms defined in the assurance case), *ArtifactPackage*s (to record artifacts referenced in the assurance case) and most importantly *ArgumentPackage*s (to caontain structured arguments for the system safety/security).

SACM provides concepts for self-contained model based assurance cases (although currently there has not been approaches and tools to achieve it) in its *Base* component shown in Figure 6. For a *ModelElement* in SACM, it can have a number of *UtilityElement*s, in this work, we particularly focus on the *ImplementationConstraint* concept, using which we describe the validation rules against engineering models. In addition, it can also be seen that a *ModelElement* can "cite" other *SACMElement* via its *CitedElement* association. This is a powerful mechanism, as it allows the users of SACM to cite any *ModelElement* contained within one model.

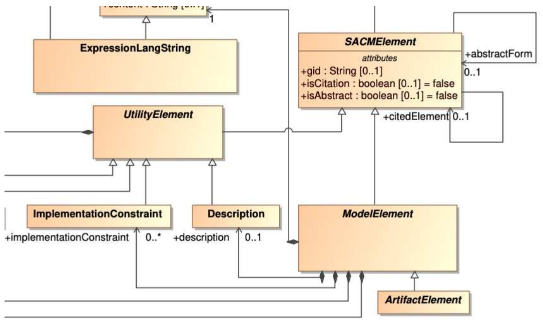


Fig. 6: The Base component of SACM [20].

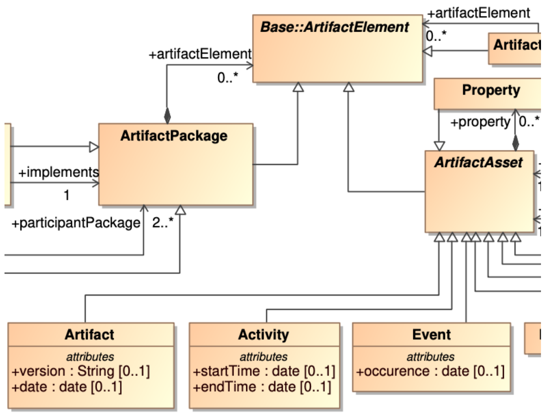


Fig. 7: The (partial) Artifact component of SACM [20].

Another SACM component worth mentioning is the *Artifact* component. , as shown in Figure 7. In this work, we make use of the *Artifact* class for demonstration purposes. It is worth mentioning that SACM concepts do not provide facilities to refer to engineering artifacts external to the assurance case. Therefore, we make use of the *Property* concept in the *Artifact* component, to create a *Property* for each *Artifact*, named "document", that record the location of an external engineering artifact (stored in the *+description* feature of the *Property* as illustrated in Figure 6). Since the engineering artifacts we refer to are meant to be models, they often have their corresponding metamodels or metadata. Thus, in the *Property*'s *ImplementationConstraint* feature, we store the location of metamodel/metadata of the "document" we refer to. In this way, we can support the referent to an engineering artifact with its metamodel/metadata (more in Section V). The above mechanism works for all sub-classes of *ArtifactAsset*.

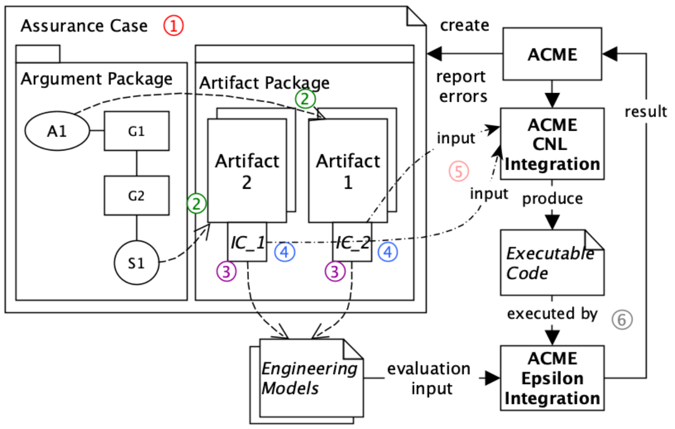


Fig. 8: Overview of the proposed approach.

We now discuss our assurance case validation approach, which is illustrated in Figure 8. There are 6 steps in our approach. In Step ①, we use ACME to create an assurance case. In this paper, we focus on the relationship between *ArgumentPackage*s and *ArtifactPackage*s. We create *ArgumentPackage*s containing arguments regarding system safety, and we create *ArtifactPackage*s containing traceability to referenced engineering artifacts/models. In Step ②, we establish links from GSN elements to elements in *ArtifactPackage*s, using the "cite" mechanism provided by SACM. In Step ③, we establish traceability links from *Artifact*s (contained in an *ArtifactPackage*) to engineering models. As discussed above, for an *Artifact*, we record the locations of an engineering model and its metamodel/metadata. In Step ④, we use SACM's *ImplementationConstraint* concept to create validation rules (*IC_1 and IC_2 in Figure 8*) for the referenced engineering models. Validation rules are used to check certain properties in the engineering models with regard to system safety (e.g. system behaviour model). For ACME, validation rules can be: 1) code written in the Epsilon Validation Language (EVL) or 2) expressions conforming to the CNL. If the validation rule is written in EVL, ACME executes it using Epsilon against the engineering model that the *Artifact* refers to, the validation result is processed and error are reported in ACME. If the validation rule is written in CNL, in Step ⑤, the rule is processed by the integrated CNL Processor, which parses CNL expressions into validation rule models, and then generate executable code (in this work we generate validation rules written in EVL). However, it is to be noted that the CNL can be transformed into any programming language. In Step ⑥, the executable code is executed by Epsilon against the engineering model (that the *Artifact* refers to), the validation result is processed by ACME and errors are reported.

IV. Implementation of CNL and generation to executable code

In this section, we discuss in detail the implementation of the CNL, inspired by the works discussed in [26], [27].

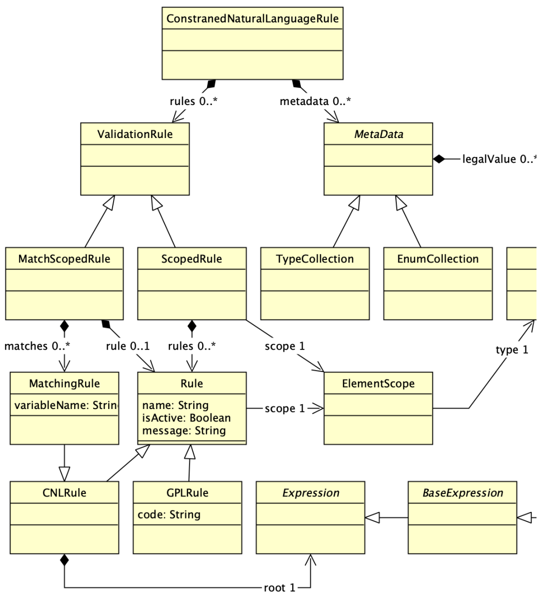


Fig. 9: A segment of the CNL metamodel.

A. CNL Metamodel

In MDE terms, the CNL metamodel captures the abstract syntax of the language, in the sense that it only contains the concepts (types) of the CNL, but not how the CNL looks like. Figure 9 shows an essential segment of the CNL metamodel. We discuss briefly these key concepts:

- *ConstrainedNaturalLanguageRule* acts as the root element, it contains a list of *ValidationRule*s (validation rules written in CNL against the model to be validated) and a list of *MetaData* (metadata/metamodel concepts of the model to be validated).
- *MetaData* is used to list meta elements inside a model to be used by the validation rules. For example, if we want to validate a state machine model, then in our *MetaData*, we need to specify *TypeCollection*, with their *legalValue*s (captured by class *MetaValue*, which is of type *BaseExpression*): State{name}, Transition{source, target}, where "State" is a *Type* and "name" is a *Feature*, same principles apply to "Transition" and its features "source" and "target". In addition, enumeration types are captured using classes *EnumCollection* and *Enum*.
- *ScopedRule* is the first type of *ValidationRule*, where one or more rules are written against a single scope (captured using *ElementScope*, which points to a *Type* in the metadata). When describing validations, a scope must always be provided. For example, consider the code segment written in Epsilon Object Language (EOL) to query a state machine model:

```
var hcm = State.all().select(sls.name = "HCM").
    first();
var transitions = Transition.all().
    select(tlt.source = hcm);
return transitions.size();
```

In this code segment, we want to query the size of *Transition*s which have their sources connected to a *State* named "HCM". We define the scope of our query, *Transition.all()*, which includes all instances of *Transition* in the state machine. The same principle applies to validation languages such as Epsilon Validation Language (EVL):

```
context Transition {
    constraint rule_1 {
        check {
            return self.source.name = "HCM"
                implies Sequence{'t4', 't5', 't6'}.
                    includes(self.name);
        }
    }
}
```

where the "context" of the validation rule is focused on all *Transition*s.

- *MatchScopedRule* is the second type of *ValidationRule*, where a single rule is written against a sub-collection of data defined in the matches (captured using *MatchingRule*). These collections can be in different scopes, allowing rules to link queries of multiple domain elements together into a single rule. In the above code example, first-order logic operations *select()* and *includes()* can be considered *MatchingRule*s.
- *CNLRule* is a rule written against elements in the metadata domain. It contains a root element (captured by the *Expression* class), which defines the rule once parsed.
- *GPLRule* is used to capture rules written in Generic Programming Language against elements in the metadata domain. It contains a String with relevant syntax conforming to the language (i.e. syntax native to the programming language). This rule allows complex expressions which may not be expressible in CNL to be written in the same document as the natural language rules.
- *Expression* is the common supertype for all expression elements a rule can be made up of. In our CNL implementation we include: *CompareisonExpression*s such as "equality", "inequality", "greater than", "less than", "greater than or equal to" and "less than or equal to"; *ArithmeticExpression*s such as addition ("+"), deduction ("-"), multiplication ("*") and division ("/"); *LogicalExpression*s such as "implies", "or", "xor", "and", "if and only if"; *DateExpression*s such as before date, after date, on or before date and on or after date.
- *BaseExpression* captures atomic expressions such as feature call expression, primitive type expressions, Enum expressions, value containment expressions, etc.

B. CNL Xtext Grammar

We now discuss the Xtext grammar which is used to automatically generate the CNL parser, the CNL text editor, and associated facilities such as content assist and syntax validation. The CNL grammar definition is made publicly available[2]. For simplicity, we discuss grammar that is relevant

[2]https://github.com/SystemsAssuranceGroup/ACMECNL/blob/58ce2b5c3959e45ac04831e7ccffbc159030765f/org.eclipse.acme.cnl/src/org/eclipse/acme/cnl/ACMECNL.xtext

to the segment of the CNL metamodel shown in Figure 9, and we refer to the segments of the grammar by identifying their line numbers in the grammar definition, the readers are invited to refer to the grammar definition alongside this paper.

The declaration of our CNL (named “ACMECNL”) is in lines 1–4. We want the Xtext parser to produce instance models of the CNL metamodel, so we import the CNL metamodel and the Ecore metamodel. We then define our own terminals in lines 11–30, we override terminals INT and ID from Xtext’s default terminals definition to 1) tell Xtext parser to generate EInt for INT terminals and 2) support more flexible IDs.

We also define *ConstrainedNaturalLanguageRule*, in line 6, it should have a number of *ScopedRule*s or *MatchScopedRule*s, and some *MetaData* instances.

The *ScopedRule* and *MatchScopedRule* are defined in lines 32 and 54. For a *ScopedRule*, the users can write CNL expressions as below:

for all State **the** name **must exist**

in which “State” is an *ElementScope* (defined in line 67) and “the name must exist“ is a *ComparisonExpression* (defined in line 67), where its left hand side expression is a *FeatureValue* (“the name”, defined in line 214) and its right hand side expression is an *ExistanceValue* (“must exist”, defined in line 271). For a *MatchScopedRule*, the users can write CNL expressions such as:

find all data in Transition
    where name **from** source **is** ’MOM’
then the name **is either** ’t4’ **or** ’t5’ **or** ’t6’ **or** ’t9’

in which *“find all data in ... where ... then ...”* is a *MatchingRule* (defined in line 61), with “Transition” an *ElementScope* (defined in line 67), *“name from source is ’MOM’ ”* a *EqualityExpression* (defined in line 132), of which the left hand side is a *VariableFeatureValue* (defined in line 217) and right hand side a *StringValue* (defined in line 229); then we specify a rule: *“the name is either ‘t4’ or ‘t5’ or ‘t6’ or ‘t9’ ”*, which is a *ValueContainment* expression (defined in line 203), in which *“the name”* is a *FeatureValue* (defined in line 214).

As previously discussed, *Rule* can be either a *CNLRUle* or *GPLRule*. *CNLRule* has a *LogicalExpression* (defined in line 91) as root, *LogicalExpression* supports logical operators *and*, *or*, *xor*, *implies* and *if and only if*, and can be used in conjunction with *ComparisonExpression*s (defined in line 93). *ComparisonExpression*s support comparison operators *equality*, *inequality*, *greater than*, *less than*, *greater than or equal to* and *less than or equal to*. *ComparisonExpression* can in turn be used in conjunction with *ArithmeticExpression* (defined in line 130), which supports arithmetic operators +, -, * and /. The above expressions need to be used with *BaseExpression*s, as defined in line 163 (we do not go into details about *BaseExpression*s).

In contrast, a *GPLRule* contains simply a String, which contains the general programming language rule (in our example, validation rules written in EOL). The purpose of the GPLRule is to avoid having tedious CNL statements when expressing nested property access. For example, to check the equality of nested property in EOL, one may write:

```
a.b.c.d.e = "hvel"
```

which is simple and easy to understand. However, written in CNL grammar, it becomes:

e **from** d **from** c **from** b **from** a **is** "hvel"

which is tedious to write and difficult to comprehend. Hence, the users have the flexibility of using *GPLRule*s in scenarios of this kind, in which CNL grammar loses its power of expressiveness.

As previously discussed, a *ConstrainedNaturalLanguageRule* should contain *MetaData* (defined in line 250), so that *ScopedRule*s or *MatchScopedRule*s can be aware of the meta information in the model to be validated. *MetaData* can be either *Types* (line 253) or *Enumerations* (line 256). *Types* can have *legalValues* of type *TypeValue* (which corresponds to *Type* in the CNL metamodel, line 262); *Enumerations* can have *legalValues* of type *Enum* (which corresponds to *Enum* in the CNL metamodel, line 259). In our case study (a state machine example), before we write any rules in CNL, we must declare meta information as follows:

types State{name}, Transition{source, target}

which corresponds to the syntax explained above, and declares that there is a *Type* called “State”, with a *Feature* called “name”, and another *Type* called “Transition”, with two *Feature*s called “source” and “target”.

The Xtext grammar is consumed by the Xtext framework, which automatically generates a CNL parser, and a text editor with syntax highlighting.

C. Executable Code Generation

With the Xtext grammar defined, we are able to obtain instance models that conform to the CNL metamodel. Our next step is to perform a model-to-text transformation, which generates executable code written in our choice of validation languages – EVL [28].

CNL is designed for validation against data in the different *backends*, which can be documents, models and databases. Sometimes there are variations of similar concepts, such as wrapping identifiers or escaping special characters. To promote the general use of the CNL for validation, we allow configurations to be made inside a *mapping model*, which is used to map CNL terms to terms in the target *backend*. Such configurations can be injected to the CNL instance model before the model-to-text transformation.

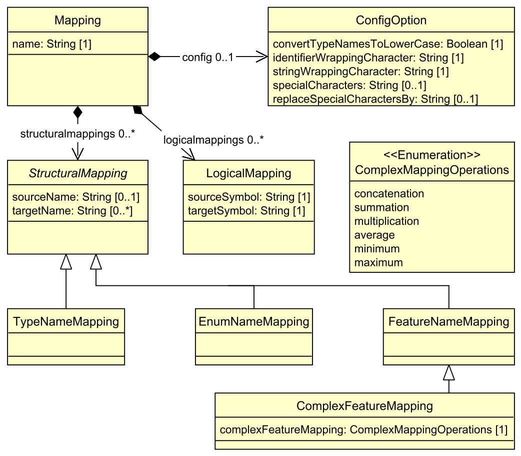


Fig. 10: The Mapping metamodel.

Figure 10 shows the mapping metamodel. Class *Mapping* is the container for the mapping from CNL to a target *backend*. In *ConfigOption*, one can specify whether type names need to be converted to lower case, wrapper character for wrapping identifiers, wrapping character for String values, special characters in the CNL and characters to replace such special characters in the target *backend*. A mapping model contains a number of *StructuralMapping*s, which essentially map names of types, enumerations and features into their correspondences in the target *backend*. For example, in CNL, identifiers have to be one continuous String. For a feature, named “firstName”, it may map to a column name in a relational data based called “first name”. In this situation, we may use a *FeatureNameMapping* to map “firstName” (*sourceName* in the *FeatureNameMapping*) in CNL to “first name” (*targetName* in the *FeatureNameMapping*)in the relational database. *TypeNameMapping*, *EnumNameMapping* and *FeatureNameMapping* can be considered as 1 to 1 mappings (we perform model validation on the mapping model to enforce that the cardinality of feature “targetName” for these classes are 1), we also support 1 to n mappings through *ComplexFeatureMapping*. *ComplexFeatureMapping* is designed to enable the accesses to multiple features in the *backend* through one feature in CNL. For example, in CNL we can define a *Feature* called “fullName”, in the mapping model, we can define a *ComplexFeatureMapping* and state that we want to map “first name” and “last name” in the target *backend*, with *complexMappingOperation* to be “concatenation”, by doing this, we can validate against one *Feature* defined in CNL but actually validate against two features in the target *backend*. We can perform numerical *ComplexMappingOperation*s such as “summation” and “multiplication”, as well as comparative *ComplexMappingOperation*s such as “average”, “minimum” and “maximum” on features that are of numerical types or collection types. At last, different *backends* may use different symbols for denoting logical operators such as equality, negation, etc., hence it is necessary to perform a mapping on logical operators (for example EVL uses = instead of == for equality).

With the defined *mapping model*, we perform an model transformation to migrate the declared CNL type, feature and enumeration names into their mapped names[3]. Then, we perform a model-to-text transformation, written in the Epsilon Generation Language (EGL) [29]. Our transformation targets EVL, but the approach can be generalised to target any validation language, such as the Object Constraint Language (OCL). We provide the pseudo code for the transformation in Algorithm 1.

We provide a helper function, named *flatten()* to help turning nested CNL model elements into flat Strings. Helper function *flatten()* takes four parameters and returns a String, “ret” enables the users to insert any String before the returned result, “scopedElementVariable” is the variable name of the scoped element, “context” is the name of the scope and “scopedCollection” is the collection of model elements in the defined scope. *flatten()* produces strings based on the element that calls it (the *self* variable). For example, for a *ValueContainment* in CNL:

```
the name is either ’t4’ or ’t5’ or ’t6’ or ’t9’
```

the *flatten()* function will produce the String:

```
Sequence{’t4’, ’t4’, ’t6’, ’t9’}.includes(self.name);
```

[3]For simplicity we do not discuss this transformation in detail

Before moving on to explaining the transformation, it is necessary to discuss the syntax for EVL, shown in Listing 1.

```
1 context <name> {
2     (guard (:expression)|({statementBlock}))?
3     (invariant)*
4 }
5 constraint <name> {
6     (guard (:expression)|({statementBlock}))?
7     (check (:expression)|({statementBlock}))?
8     (message (:expression)|({statementBlock}))?
9 }
```

Listing 1: The Syntax for EVL context and invariant

In EVL, a scope can be declared by *context* (line 1) where “name” is used to specify a type (in the *backend* to be validated). In the *context*, an optional *guard* can be defined to limit the applicability of the *context* to a narrower subset of instances. A *context* may contain a number of *invariant*s which are validation rules applicable to the *context*. A *constraint* is an *invariant*, in which an optional *guard* can be defined to limit the applicability of the *constraint*, an optional *check* can be defined to contain the validation rules, which can be either an *expression* or an *statementBlock*. In addition, an optional *message* can be defined to notify information upon failed validations.

Algorithm 1: Generating executable code from CNL instance model.

```
1  for sr in {all ScopedRules} do
2      let context = sr.scope.type.name
3      output "context" + context + "{"
4      let root = sr.root
5      let name = rule.name
6      output "constraint ValidateScopedRule_" + name +"{"
7      "check { return " + root.flatten("", "self", context, null) +
         "}"
8      "message{" + rule.message == null? "no error message":
         rule.message + "}"
9  end
10 for msr in {all MatchScopedRules} do
11     let name = msr.rule.name
12     output constraint ValidateMatchedScopeRule_ + name +
         "{"
13     "check{
14     let expression = "var collection = new Sequence;"
15     let rule = msr.rule
16     let matches = msr.matches
17     for match in {mathces} do
18         let context = match.scope.type.name
19         expression+= "collection.addAll(" + context+".all.
             select(v|"+match.root.
             flatten("","v",context,null)+");"
20     end
21     expression += "for (element in collection) {"
22     expression += "var result =" +
         rule.root.flatten("","element",null,collections) + ";"
23     expression += "if (result = false) return false;}"
24     expression += "return true;";
25     output expression
26     output }
27 end
```

We generate validation rules in EVL for all *ScopedRule* and *MatchScopedRule* in Algorithm 1. The generation for *ScopedRule*s is straight forward. We generate a *context* in EVL in line 3, in which we generate a *constraint* (line 6) with the name of the *ScopedRule*, we generate a *check* (line 7), in which we obtain a flattened String by calling the *flatten()* function, we generate a *message* in which we preserve the massage from the *ScopedRule*.

For *MatchScopedRule*s, we are creating a “contextless” *constraint* for each *MatchScopedRule* (line 12), since we are aggregating the collections of elements in the *backend* through *MatchingRule*s (note that a *MatchScopedRule* can have a number of *MatchingRule*s which may not be in the same scope). Thus, we generate a *Sequence* (line 14) to contain the elements for the *MatchingRule*s. We iterate through each *MatchingRule* in line 17–20, and generate texts to ask EVL to collect model elements based on the *MatchingRule*. Finally, we generate text to iterate through all elements in the *Sequence* (line 21–24), and perform validation against each of the elements, false is returned if any validation fails (line 23), we output the text in line 25. With the transformation, the CNL *ScopedRule*:

```
for all Transition the source is not empty and
    the target is not empty
```

Listing 2: Example CNL rule 1.

will be transformed into:

```
context Transition {
  constraint ValidateRule_0{
    check{return ‘Model‘.isPropertySet(self,’source’) and
          ‘Model‘.isPropertySet(self,’target’);}
    message{return "no error message defined for rule: 0";}
  }
}
```

The CNL *MatchScopedRule*:

```
find all data in Transition where name from source
    is "MOM" and name from target is "HCM"
        then name is either "t4"or "t5" or "t6" or "t9"
```

Listing 3: Example CNL rule 2.

will be transformed into:

```
constraint ValidateMatchingRule_0 {
  check{
        var collection = new Sequence;
        collection.addAll(Transition.all
              .select(v|v.source.name == ’MOM’ and
                  v.target.name == ’HCM’));
        for (element in collection) {
              var result = Sequence{’t4’,’t5’,’t6’,’t9’}
                  .includes(element.name);
              if (result = false)
              return false;
        }
        return true;
    }
    message{ return "no error message defined for rule: 0"; }
}
```

The CNL metamodel, CNL grammar and model transformations come together as shown in Figure 11. CNL metamodel and CNL Xtext grammar are consumed by the *Xtext engine* (①), which automatically generates a *CNL parser* and a *CNL editor* (②). Xtext allows the definition of *Generator* (used to generate artifacts from parsed Xtext grammar), in which we define 1) endogenous transformation (based on the *mapping model*) to migrate CNL names (for *Type*s, *Enum*s and *Feature*s) to target *backend* names; and 2) model-to-text transformation to generate EVL validation rules from CNL instance models (③). CNL rules are created using the CNL editor, which parses the rule and produces CNL model (④). With the CNL model and the mapping model as inputs to the *Generator*, it generates

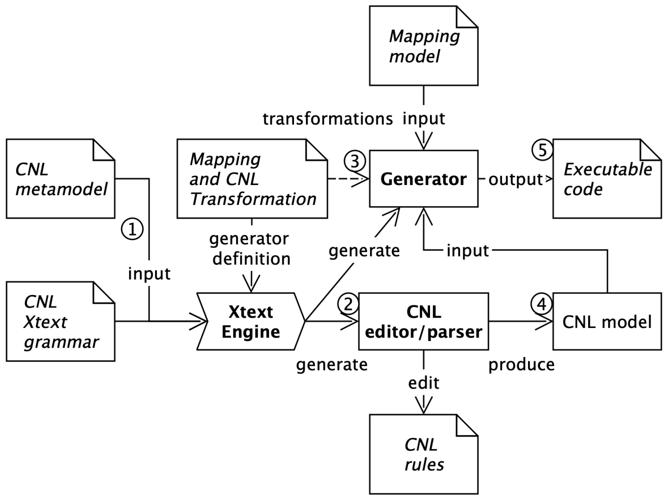


Fig. 11: CNL implementation and executable code generation.

the executable code (in our case the EVL code) (⑤). To run the validation rule against a *backend*, one needs to setup a run configuration (e.g. for EVL), specify the *backend* document (e.g. model) and its metadata (e.g. metamodel) and execute the validation rules against the *backend*.

V. Integration with ACME and the AUV Case Study

In this section, we discuss the integration of the CNL implementation into ACME. We discuss the approach of automatic assurance case validation and apply our approach on an assurance case developed for an Autonomous Underwater Vehicle (AUV). The assurance case is developed based on a RoboChart [30] model, from which we develop a modular assurance case using ACME, and apply our approach for traceability and automated evaluation with CNL.

The AUV is a portable untethered Remotely Operated Vehicle, equipped with a visual mapping system and verified on-board autonomy. The aim is to make it capable of conducting light intervention tasks, such as oil and gas surveys and offshore coring. Industrial partners for the AUV engage with regulators through ongoing contributions to autonomy regulatory work to ensure regulatory compliance. Thus, the use of a structured assurance case is vital to communicate the evidence of safe operation to non-specialists, especially in the aspect of software controlled autonomous behaviour.

In this paper, we focus on the assurance case for the *Last Response Engine* (LRE), which provides runtime safety assurance. The AUV can either be under operator control, or running autonomously. If operating autonomously, the responsibility for satisfying the safety requirements lies with the LRE, which can engage evasive manoeuvres if necessary.

The overall architecture of the AUV is modelled in Figure 12 using the RoboChart language. The robotic platform (*AUV_Platform*) acts as an abstraction layer for the hardware, and provides access to sensors and actuators. The LRE (*LRE_Ctrl*) sits between the operator (*AUV_Operator*) and the autopilot component (*AUV_Autopilot*). The operator, which can be either a human or navigation system, provides instructions to the LRE to support execution of tasks, such as requesting a particular heading and velocity. The autopilot

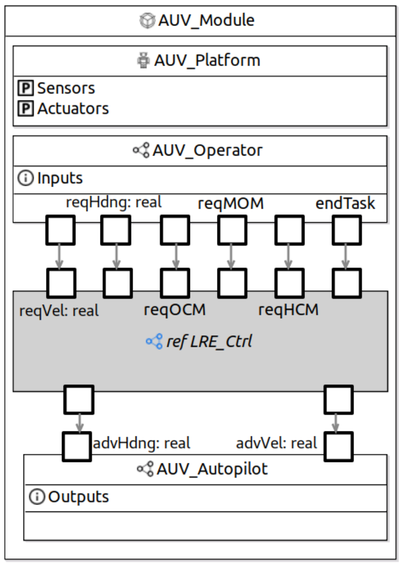


Fig. 12: Overall Architecture of the AUV and LRE.

controls the AUV actuators, and takes advice only from the LRE.

The LRE functions in four modes: Operator Control Model (OCM), Main Operating Mode (MOM), High Caution Mode (HCM) and Collision Avoidance Mode (CAM). Whilst in OCM, the LRE passes through control inputs from the operator to the autopilot. MOM is where the LRE takes control for normal behaviour at maximum speed. HCM is for the situation when the AUV is getting close to an obstacle, and so the LRE lowers the velocity. Finally, CAM is the mode where a potential unavoidable collision has been detected, and the AUV is manoeuvring away from the obstacle.

The LRE keeps an *obstacle register*, which stores identified obstacles, through sensor readings. In each behavioural cycle, the LRE calculates the closest obstacle and determines whether it should apply evasive manoeuvres or switch into high caution mode (HCM).

The LRE state machine is shown in Figure 13. It implements the LRE's behavioural requirements and specifies the conditions on switching to different operation modes. We are particularly interested in the *Transition*s from MOM to HCM (i.e. 't4', 't5', 't6' and 't9'). 't4' is triggered when hvel >= 0.1 (horizontal velocity of the AUV is greater than or equal to 0.1 m/s) and *hdist(cstc)* (horizontal distance of a *cstc* (closest static obstacle)) is shorter or equal to *StaticObsHorizDist* (shortest distance allowed to an obstacle horizontally), 't5' is triggered when vvel >= 0.1 (vertical velocity of the AUV is greater than or equal to 0.1 m/s) and *vdist(cstc)* (vertical distance of a *cstc*) is shorter or equal to *StaticObsVertDist* (shortest distance allowed to an obstacle vertically), 't6' is triggered when *vdist()* (vertical distance to an obstacle) is less than or equal to *StaticObsDfltVertDist* (default distance allowed to an obstacle vertically), and finally 't9' is triggered upon *reqHCM* (the LRE requests HCM model).

A. Assurance case for the AUV

With the defined behavioural model for the AUV, we discuss the assurance case (created using ACME) for the AUV, and focus on relevant parts of it for the LRE. We focus on the scenario of static obstacle avoidance for the LRE, the safety argument fragment of which is shown in Figure 14. The top level *Goal C6_a* states that upon detecting a close static obstacle, LRE should advise the autopilot to switch to HCM and reduce the velocity of the AUV to 0.1m/s. *C6_a* is in the context of, and thus contingent upon, *Assumption LRE_A1*, and *Away Goal*s *Autopilot* and *Sensors*[4]. *LRE_A1* ensures that the argument need only hold when the operator is not in control; the alternative case is handled by the *Operator* module. We support *C6_a* by decomposition. *Strategy LRE_S1* states our argument strategy, which is in the context of *Context LRE_C1*.

We focus on *Goal C7_a* for illustration. In *C7_a*, we state that the LRE should activate HCM if there are potential collision risks. We support this *Goal* with *Solution Sn1*, which states that transitions to HCM mode from MOM should be modelled by the behavioural model in Figure 13.

B. Traceability to AUV Models and Evaluation

Certain GSN elements, such as *Context*s and *Solution*s, can refer to models/documents external to the assurance case. With traditional GSN approaches, references to external models/documents are informal (only names of the models/documents are referenced and tracing is performed manually) and their validation is often performed manually. In ACME, we provide this traceability support by using SACM's *ArtifactPackage*s (as discussed in Section III) and the ability to automatically validate linked artifacts.

We illustrate traceability with *C7_a*, which is supported by several transitions in the RoboChart state machine. To be able to reference elements of the RoboChart model shown in Figure 13, we create an *Artifact* named *LRE_HCM_R1*, which will be used by solution *Sn1*. In ACME, elements can be edited using property dialog windows. The properties of *LRE_HCM_R1* are shown in Figure 15. As discussed in Section III, for elements in the *ArtifactPackage*, it is possible to attach *Property*s to them. Therefore, we make use of this feature to store locations of models/documents external to the assurance case. In ACME, we add a default *Property* for each *Artifact*, and we record "document" (which stores the location of the document/model), and "metadata" (which stores the location of the metadata/metamodel)[5], the users can browse and select their document/madatada pairs external to their assurance cases in the "References" section of the dialog in Figure 15.

C. Validation of traced model and integration of CNL into ACME

With our established way to refer to external models/documents, our next step is to look into making use of such models/documents within the assurance case. As discussed in Section III, we use *ImplementationConstraint*s to store model validation rules in *Artifact*s. We support validation rules written in the Epsilon Object Language (EOL) [24] and rules written in CNL. For CNL rules, we integrate the CNL editor in our property dialog. But instead of relying on Xtext's *Generator*, we obtain the CNL instance model directly, and perform the model-to-text transformation from CNL to EVL programmatically. Since we are evaluating against models

[4]The away goals must be supported in the *Platform* and *Autopilot* GSN modules for the LRE module to be valid.

[5]It is to be noted that engineering artifacts should be organised in the same project where the assurance resides.

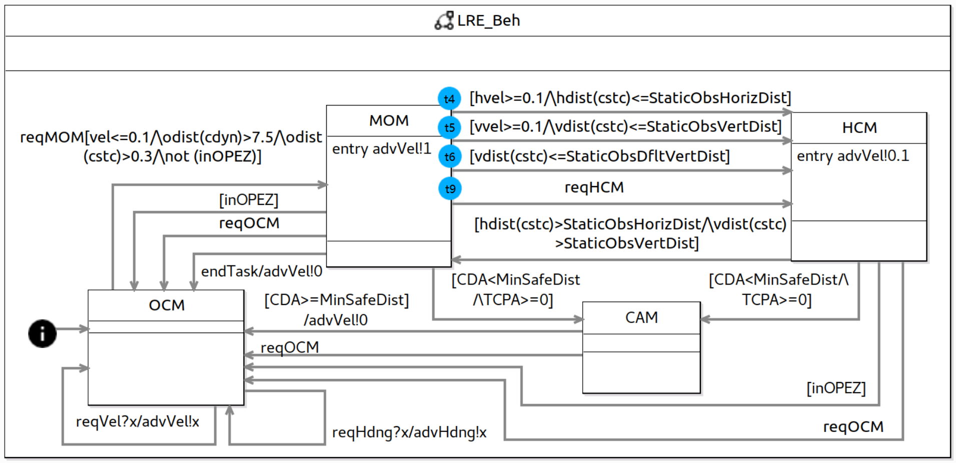

LRE_Beh
t4 [hvel>=0.1/\hdist(cstc)<=StaticObsHorizDist]
t5 [vvel>=0.1/\vdist(cstc)<=StaticObsVertDist]
t6 [vdist(cstc)<=StaticObsDfltVertDist]
t9 reqHCM
MOM
entry advVel!1
HCM
entry advVel!0.1
reqMOM[vel<=0.1/\odist(cdyn)>7.5/\odist (cstc)>0.3/\not (inOPEZ)]
[inOPEZ]
reqOCM
endTask/advVel!0
[hdist(cstc)>StaticObsHorizDist/\vdist(cstc) >StaticObsVertDist]
[CDA<MinSafeDist /\TCPA>=0]
[CDA<MinSafeDist/\ TCPA>=0]
OCM
[CDA>=MinSafeDist] /advVel!0
CAM
reqOCM
[inOPEZ]
reqVel?x/advVel!x
reqHdng?x/advHdng!x
reqOCM

Fig. 13: LRE RoboChart State Machine

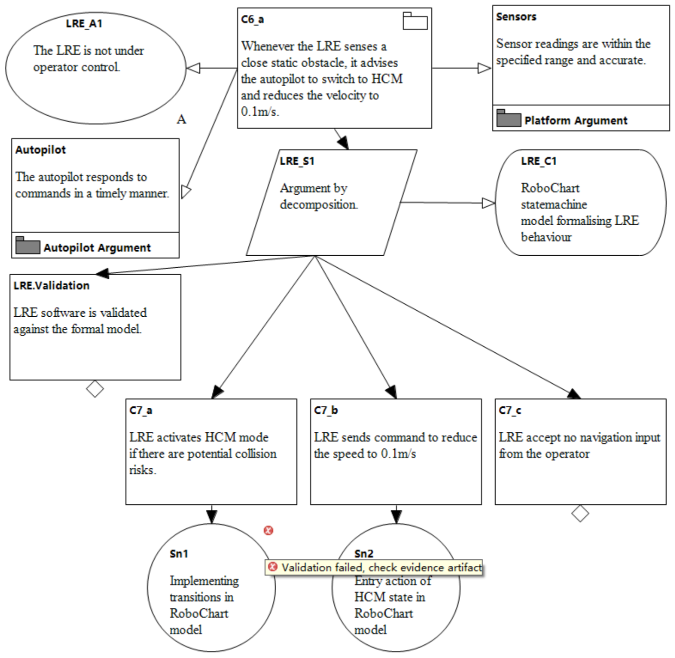
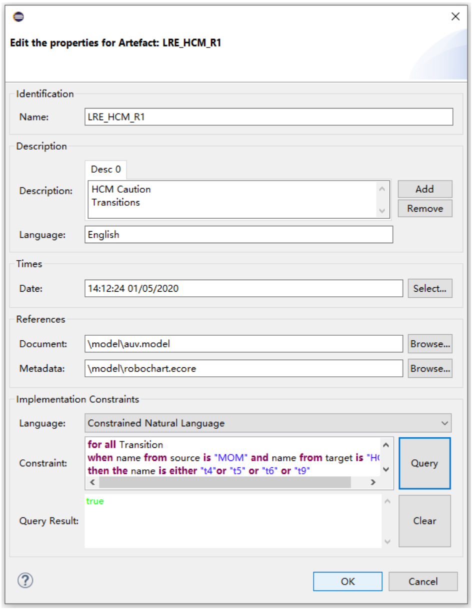

LRE_A1
The LRE is not under operator control.
A
C6_a
Whenever the LRE senses a close static obstacle, it advises the autopilot to switch to HCM and reduces the velocity to 0.1m/s.
Sensors
Sensor readings are within the specified range and accurate.
Platform Argument
Autopilot
The autopilot responds to commands in a timely manner.
Autopilot Argument
LRE_S1
Argument by decomposition.
LRE_C1
RoboChart statemachine model formalising LRE behaviour
LRE.Validation
LRE software is validated against the formal model.
C7_a
LRE activates HCM mode if there are potential collision risks.
C7_b
LRE sends command to reduce the speed to 0.1m/s
C7_c
LRE accept no navigation input from the operator
Sn1
Implementing transitions in RoboChart model
Sn2
Entry action of HCM state in RoboChart model
Validation failed, check evidence artifact

Fig. 14: LRE assurance case module for obstacle avoidance.

directly in ACME, no mapping is needed. Remember that in our CNL syntax, before a *Type*, *Enum* or *Feature* is used, it needs to be declared in the editor first. In ACME, since we obtain the model and the metamodel through the "Reference" section of the property dialog, we automatically inject names of all *Types* *Enums* and *Features* in the editor, so that the users can start writing CNL rules without worrying about declaring names.

In the Artifact dialog shown in Figure 15, in the "Implementation Constraint" section, we attach the validation rule in Listing 3 in Section IV. In this rule, we obtain all *Transitions* from "MOM" to "HCM" in the state machine in Figure 13. We check that the names of the *Transitions* are either 't4' or 't5' or 't6' or 't9'. This is due to the fact that in the LRE development process, unique global IDs for elements are used, if for example, *Transition* "t4" is deleted, the name "t4" cannot be used any more. In addition, *LRE_HCM_R1* depends on four more *Artifacts*:



Identification
Name: LRE_HCM_R1
Description
Desc 0
Description: HCM Caution Transitions
Language: English
Times
Date: 14:12:24 01/05/2020
References
Document: \model\auv.model
Metadata: \model\robochart.ecore
Implementation Constraints
Language: Constrained Natural Language

Constraint:
```
for all Transition
when name from source is "MOM" and name from target is "H
then the name is either "t4"or "t5" or "t6" or "t9"
```
Query Result:
```
true
```

Fig. 15: ACME dialog to edit properties of an Artifact.

LRE_HCM_R1_t4, *LRE_HCM_R1_t5*, *LRE_HCM_R1_t6* and *LRE_HCM_R1_t9*. In *LRE_HCM_R1_t4*, we specify rule

```
find all data in Transition where name = "t4"
then
<EOLExpression>
return condition.left.left.name = "hvel" and
        condition.left.right.value = 0.1 and
      condition.right.left.name = "hdst" and
      condition.right.left.args.first = "cstc" and
      condition.right.name = "StaticObsHorizDist";
</EOLExpression>
```

to check the *condition* of 't4' (shown in Figure 13). In here we make use of the *GPLRule* in which we incorporate EOL expressions in the "<EOLExpression>" tag. This is because that the RoboChart metamodel is rather complex, and the condition

$$[hvel \geq 0.1 \wedge hdist(cstc) \leq StaticObsHorizDist]$$

in the RoboChart model is rather complex, such that if CNL is used, instead of writing

```
condition.left.left.name = "hvel"
```

we would write nested property access expressions such as

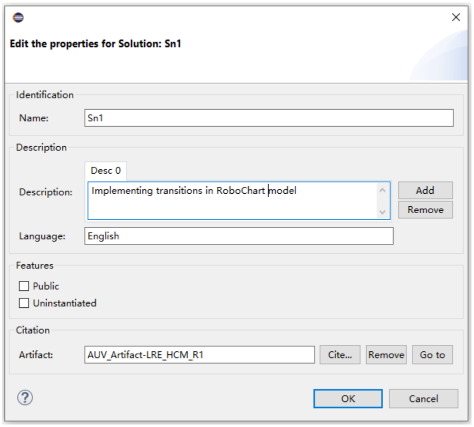
```
name from left from left from condition is "hvel"
```

which is currently not supported (to avoid incomprehensible statements of this sort) This is an example of when verboseness of the CNL is adversarial. However, the use of *GPLRule* addresses this problem efficiently. Artifacts *LRE_HCM_R1_t5*, *LRE_HCM_R1_t6* and *LRE_HCM_R1_t9* have similar validation rules.

The user can evaluate the query inside the dialog by pressing the "Query" button. ACME will load the model specified in the "Reference" section and transform-execute the validation rule, the result of which is displayed in the "Query Result" text field. It is to be noted that any *Artifact*s that *LRE_HCM_R1* will also be validated, and the results will be displayed in the "Query Result" text field. It is important to note that since Epsilon supports arbitrary modelling technologies, in theory in ACME we can support models defined using other technologies such as Simulink, and PTC Integrity Modeller.

D. Automated validation of the assurance case

With trace and validation rule defined, *Artifact LRE_HCM_R1* can then be used as supporting evidence for our assurance case (*Solution Sn1*). To do this, within *Sn1* we "cite" *LRE_HCM_R1* (defined in the *AUV_Artifact* package) in the "Citation" section of the property dialog, shown in Figure 16.


Edit the properties for Solution: Sn1

Identification

Name: Sn1

Description

Desc 0

Description: Implementing transitions in RoboChart model

Language: English

Features

☐ Public

☐ Uninstantiated

Citation

Artifact: AUV_Artifact-LRE_HCM_R1

Fig. 16: ACME dialog to edit *Solution Sn1*.

With *Artifact*s "cited" by GSN elements, it is possible to validate the assurance case from a single point. For this purpose, we create a context menu entry "validate" in GSN editors. When we validate a GSN module in ACME, it automatically traces to *Artifact*s from either *Solution*s or *Context*s; then, ACME executes model validation rules in the *Artifact*s written in either EOL or CNL. If there are problems (model validation returning "false"), ACME puts an error marker on the offending element, as shown in Figure 14.

This process of validation can be performed at regular intervals to ensure that updates to models and other engineering artifacts do not invalidate the assurance case. For example, if a developer removed one of the transitions from MOM to OCM, ACME will be able to detect this change and flag an error. In this way, we can automate the validation of heterogeneous evidence in an assurance case. In addition, we can proactively manage changes in the engineering artifacts from the perspective of the assurance case.

VI. Evaluation

In this section, we present the empirical results obtained when evaluating the CNL integration into ACME for the AUV case study.

A. Coverage

We first classify the rules (used in the AUV case study) into 8 categories and then analyse the coverage of the CNL with respect to the total number of rules. These rules are provided originally in a mixture of natural language and pseudo codes, and are written so that domain experts can understand them, hence are not amenable to machine consumption in any way. In order for these rules to be executed against the AUV model, they have to be understood by a domain expert and then a technical expert will have to write the appropriate low-level (EVL) code representing these rules. This process requires stakeholders with different expertise to collaborate and can introduce further risk as the two interpretation steps need to be in line with one another.

Category	Count
type/enum check	155
comparison	32
comparison + logic	7
using variables	155
duplicate check	2
multi-key match	11
enumeration sub-matching	3
complex rule	7
	372

TABLE I: Classification of queries.

Table I presents these categories. We observe that the large majority of rules fall under either simple type/enumeration checks or elaborate rules requiring the use of variables and functions (matching rules), often across different domain elements. From the remaining rules, the 7 complex rules are noteworthy as it was decided that attempting to further classify them or convert them to CNL was not efficient. Instead, these rules are flagged as complex and meant to be executed through the use of GPL rules that are written in the target language used to execute against the data itself. Finally, the category of enumeration sub-matching is not yet supported, even though such a feature can be added in further iterations of the tool, as mentioned in Section 8. As such, we achieve 98% coverage (as we do not currently offer CNL expressiveness for complex rules and enumeration sub-matching rules), whilst opening the possibility (through the use of GPL rules) for any rule to be written in the CNL document regardless, in order to ensure that a single document contains all the rules that need to be executed, regardless of whether they can be actually expressed in CNL.

B. Verboseness

We then evaluate the verboseness of the rules written in CNL. Should the CNL form of the rules be disproportionate to the complexity of the rules (the size of the rule written in the execution language) then it may be unreasonable to expect them to be written by domain experts as it will become tedious to write extremely long CNL rules. As such, we compare the size in characters (ignoring whitespaces) of various rules written in CNL with the rule written in EVL, as a representative sample of verboseness.

Category	CNL	EVL
type/enum check	33	86
comparison	51	118
comparison + logic	135	270
using variables	308	523
duplicate check	55	226
multi-key match	63	485

TABLE II: Rule character count ignoring spaces.

Table II shows the relevant character count for a representative rule written for each of the categories the tool supports. It can be seen that CNL rules are much less verbose than EVL constraints written in Epsilon. Considering the fact that the EVL programs are generated by the tool and as such attempt to be as minimal as possible (as they do not care about human readability at all), we have gained confidence that writing rules in CNL require less effort than the same rule written by the relevant expert in EVL or any other validation languages.

C. Extensibility

Extensibility shall be considered in three aspects. First, **language extension** can be achieved by simply modifying the CNL grammar and re-configure the CNL parser to extend expressions/semantics for intended use. Regarding extending the subset of English supported by the CNL, this would require adding the new expressions in the Xtext parser that reads the CNL document and creates the relevant model. Since adding new English phrases is unlikely to affect the model itself but rather the parser, we believe that the CNL framework is extensible in this regard as only one component of the system needs to be adapted to add this functionality. We attempted 10 re-configurations by adding new expressions in the Xtext parser, and the result confirms our intuition with regard to language extension. Secondly, **domain extension** is naturally supported since CNL does not target a specific domain with the help of the mapping model. Finally, **execution technology extension** is supported by the design of our approach, in the sense that any data storage technology can be supported, and the target validation language is not bound to EVL alone. Hence, any validation execution technology is supported by creating a transformation which targets the intended validation language.

D. Efficiency of CNL with ACME

We then evaluate the efficiency of CNL integrated with ACME in the AUV assurance case. Two test subjects are invited for an experiment, A (unfamiliar with Epsilon) and B (experienced with Epsilon). Test subject A is a professional in safety case development and review (in GSN) in the aviation industry, and has extensive experience in hardware design and software verification. But, test subject A has limited knowledge of modelling and model management. Test subject B is a professional in the automotive industry (certified ISO-26262 engineer), and have a basic understanding of safety case and have seen safety cases developed in GSN. Test subject B has plenty of experience in modelling and model management. CNL is explained briefly to both A and B, and a brief trainings on Epsilon is provided to A. There are two workloads WL1 and WL2, WL1 involves validating two GSN solutions (trace to and validate two different models), where WL2 involves validating a GSN module with 2 *Context*s and 3 *Solution*s (trace and validate four different models). Each test subject A and B are asked to complete WL1 and WL2 with CNL and Epsilon, respectively.

Test subject	Language	WL1 (minutes)	WL2 (minutes)
A	CNL	2.5	5.5
A	EVL	40	80
B	CNL	2	3.5
B	EVL	25	58

TABLE III: Efficiency experiment.

As shown in the table III, the time it takes for both A and B to complete the task using CNL is much less than using EVL. Test subject A takes more time to complete the tasks than B using EVL due to the limited knowledge on modeling and EVL. Test subject B take less time to complete the tasks than test subject A using both EVL and CNL, as he/she is more familiar with modelling and model validation languages. Test subject B claims that he/she prefers the mixture of CNL and EVL as the verboseness of CNL for complex models can be redundant to write.

Student	CNL(minutes)	EVL(minutes)
1	35	330
2	38	400
3	40	320
4	32	290
5	33	300

TABLE IV: Normalised efficiency experiment.

To normalise our findings, we expand the evaluation for efficiency to a group of test subjects with similar expertise. We ask 5 MSc students with an (more or less) equal amount of training on safety cases and hardware design to construct a partial assurance case for a sensor power supply unit for an AUV. We provide the hardware design model in Matlab/Simulink to the students and ask them to perform Failure Mode and Effect Analysis (FMEA) on the design and store the analysis results in an Excel spreadsheet. We then ask them to identify the safety goals for the power supply unit and construct an assurance case with the identified safety goals. We ask that they shall create at least 1 *Context* in GSN and 2 *Solution*s in the assurance case, and they shall refer to the Simulink model and the FMEA result in their assurance case. We then ask them to create validation rules in CNL and Epsilon respectively in the GSN elements so that they can refer to the Simulink model and the Excel spreadsheet and validate them within their assurance case. Again, training on CNL and Epsilon are provided to the test subjects. The time it takes to write validation rules and execute them for complete assurance case validation for each test subject is shown in Table IV.

In summary, validating assurance cases with CNL in ACME boosts the efficiency of validation, up to a factor of 10 comparing to validation with Epsilon. In addition, we also note that all experiment participants found the *executable* traceability link from the model-based assurance case and engineering models promotes the comprehensibility of the assurance case significantly, in the sense that the participants do not need to manually find the referenced engineering models, and perform validations on the models separately.

VII. Related Work

There are a number of assurance case tools that adopt MDE, such as AdvoCATE [10], ASCE [31] and CertWare

[32]. Among them, only AdvoCATE provides the support to express the traceability from an assurance case to its supporting documents. It provides "hyperlinks" to external models/documents to provide traceability, however, no means for validating the referenced engineering artifacts has been provided. AdvoCATE also provides a query language to check the well-formedness of the assurance case, it resembles model query languages such as OCL and EOL, however, its syntax dictates that the users need to spend some time to learn the query language before using it. In [12], the authors use a *weaving model* to link system models for automated assurance case instantiation. Rushby conceived of an evidential tool bus [33] that would allow integration of various verification tools to provide evidence to an assurance case, an idea that was later realised by Cruanes et al. [34]. In [35], the authors propose the use of formal languages to express assurance cases using Isabelle/SACM. Whilst the validation of Isabelle/SACM is performed by the theorem prover on Isabelle server to show the logical integrity of the assurance case, there currently is no way for the formal assurance case to trace to and validate engineering models external to the assurance case. In addition, theorem proving takes a significant amount of time, which renders this approach infeasible for the validation of assurance cases at runtime. For CNL, [26] discussed an approach to map-controlled natural language to business rules that align with Semantics of Business Vocabulary and Business Rules standard (SBVR); in [36] the authors presented a mapping from expressions written in SBVR to Drools. As good as SBVR is at introducing structure to constrained natural languages, its inherent complexity means that for smaller dialects the overhead of the standard may overshadow its usefulness.

VIII. Summary and Future Work

In this paper, we presented our approach for automated model based assurance case validation and management. With our implementation of SACM and its tool support ACME, we are able to provide a systematic approach to managing engineering artifacts (models and/or structured documents) within an assurance case, with the traceability support to the granularity at the model element level. For engineering models, we provide support which enables practitioners to attach model validation rules to SACM elements, which are automatically executed when the assurance case is validated. We point out the challenges in using model validation languages for system assurance practitioners adopting MDE, and we propose a solution to address such challenges through the use of CNL. We presented our preliminary CNL metamodel, CNL grammar and the model-to-text transformation from validation rules written in CNL to validation rules written in EVL. We then discussed the integration of our CNL framework into ACME to support the automated validation of assurance cases using CNL. This work promises that system assurance practitioners adopting MDE do not need extensive training for MDE programming languages, and the comprehensibility of traceability to models within an assurance case is promoted.

In this work, we only incorporate CNL in validation rules for engineering artifacts. In the future, we plan to support CNL in the description of SACM elements, so that we may 1) define terms and expressions used in an assurance case using SACM's *Terminology* component; 2) relate elements in *Terminology* packages within the arguments of the assurance case to promote consistency; 3) translating CNL descriptions into formal notations for machine checking the logical integrity of assurance cases. The use of CNL in arguments of an assurance case provides the possibility to promote standardisation of the language used in assurance cases, therefore provides a solid foundation for more automated operations on such completely model based assurance cases. We also plan to support validation of engineering models defined in other modelling technologies such as Simulink and PTC Integrity Modeller.

Acknowledgment

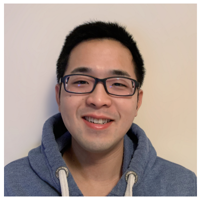
The authors would like to thank the editors for the organization of the paper review, and anonymous reviewers for their helpful feedback. This work was supported in part by the National Natural Science Foundation of China (Grant No. U1913201, U22B2041).

References

[1] EUROCONTROL. *European Organisation for the Safety of Air Navigation: Safety Case Development Manual.* 2006.
[2] IAEA. *International Atomic Energy Agency Safety Glossary: Terminology Used in Nuclear Safety and Radiation Protection.* 2008.
[3] ISO. *ISO 26262: Road Vehicles - Functional Safety.* 2018.
[4] U.K. Rail Safety Standards Board. *Engineering Safety Management Issue 4.* 2007.
[5] U.K. Ministry of Defence. *Safety Management Requirements for Defence Systems.* 2007.
[6] Mario Trapp, Daniel Schneider, and Peter Liggesmeyer. A safety roadmap to cyber-physical systems. In *Perspectives on the future of software engineering*, pages 81–94. Springer, 2013.
[7] Tim Kelly and Rob Weaver. The goal structuring notation–a safety argument notation. In *Proceedings of the dependable systems and networks 2004 workshop on assurance cases*, page 6. Citeseer, 2004.
[8] Peter Bishop and Robin Bloomfield. A methodology for safety case development. In *Safety and Reliability*, 2000.
[9] Ran Wei, Tim P Kelly, Xiaotian Dai, Shuai Zhao, and Richard Hawkins. Model based system assurance using the structured assurance case metamodel. *Journal of Systems and Software*, 154:211–233, 2019.
[10] Ewen Denney and Ganesh Pai. Tool support for assurance case development. *Automated Software Engineering*, pages 1–65, 2017.
[11] Richard David Hawkins et al. The need for a weaving model in assurance case automation. *Ada User Journal*, 2015.
[12] Richard Hawkins, Ibrahim Habli, Dimitris Kolovos, Richard Paige, and Tim Kelly. Weaving an assurance case from design: a model-based approach. In *High Assurance Systems Engineering (HASE), 2015 IEEE 16th International Symposium on*, pages 110–117. IEEE, 2015.
[13] Zhe Jiang, Shuai Zhao, Ran Wei, Dawei Yang, Richard Paterson, Nan Guan, Yan Zhuang, and Neil C Audsley. Bridging the pragmatic gaps for mixed-criticality systems in the automotive industry. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(4):1116–1129, 2021.
[14] Zhe Jiang, Shuai Zhao, Pan Dong, Dawei Yang, Ran Wei, Nan Guan, and Neil Audsley. Re-thinking mixed-criticality architecture for automotive industry. pages 510–517, 2020.
[15] Zhe Jiang, Neil C Audsley, and Pan Dong. Bluevisor: A scalable real-time hardware hypervisor for many-core embedded systems. pages 75–84, 2018.
[16] Sanjit A Seshia, Shiyan Hu, Wenchao Li, and Qi Zhu. Design automation of cyber-physical systems: Challenges, advances, and opportunities. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(9):1421–1434, 2016.
[17] Ran Wei, Tim P Kelly, Richard Hawkins, and Eric Armengaud. Deis: Dependability engineering innovation for cyber-physical systems. In *Federation of International Conferences on Software Technologies: Applications and Foundations*, pages 409–416. Springer, 2017.
[18] Ran Wei, Jan Reich, Tim Kelly, and Simos Gerasimou. On the transition from design time to runtime model-based assurance cases. In *13th International Workshop on Models@Runtime, ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2018)*, 2018.
[19] Erfan Asaadi, Ewen Denney, Jonathan Menzies, Ganesh J Pai, and Dimo Petroff. Dynamic assurance cases: a pathway to trusted autonomy. *Computer*, 53(12):35–46, 2020.
[20] Object Management Group. Structured Assurance Case Metamodel. https://www.omg.org/spec/SACM, 2019. Accessed 6th June, 2020.

[21] Ran Wei, Zhe Jiang, Xiaoran Guo, Haitao Mei, Athanasios Zolotas, and Tim Kelly. Designing critical systems with iterative automated safety analysis. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 181–186, 2022.
[22] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.
[23] Eclipse Foundation. Eclipse Modelling Framework (GMF). https://www.eclipse.org/modeling/gmp/, 2020.
[24] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. The epsilon transformation language. In *International Conference on Theory and Practice of Model Transformations*, pages 46–60. Springer, 2008.
[25] Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.
[26] Paul Brillant Feuto Njonko, Sylviane Cardey, Peter Greenfield, and Walid El Abed. Rulecnl: A controlled natural language for business rule specifications. In *International Workshop on Controlled Natural Language*, pages 66–77. Springer, 2014.
[27] Konstantinos Barmpis, Dimitrios Kolovos, and Justin Hingorani. Towards a framework for writing executable natural language rules. In *European Conference on Modelling Foundations and Applications*, pages 251–263. Springer, 2018.
[28] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. On the evolution of ocl for capturing structural constraints in modelling languages. In *Rigorous Methods for Software Construction and Analysis*, pages 204–218. Springer, 2009.
[29] Louis M Rose, Richard F Paige, Dimitrios S Kolovos, and Fiona AC Polack. The epsilon generation language. In *European Conference on Model Driven Architecture-Foundations and Applications*, 2008.
[30] Alvaro Miyazawa, Pedro Ribeiro, Wei Li, ALC Cavalcanti, Jon Timmis, and JCP Woodcock. Robochart: a state-machine notation for modelling and verification of mobile and autonomous robots. *Tech. Rep.*, 2016.
[31] Kateryna Netkachova et al. Tool support for assurance case building blocks. In *International Conference on Computer Safety, Reliability, and Security*, pages 62–71, 2014.
[32] Matthew R Barry. Certware: A workbench for safety case production and analysis. In *Aerospace Conference, 2011 IEEE*, 2011.
[33] J. Rushby. An evidential tool bus. In *Formal Methods and Software Engineering (ICFEM)*, volume 3785 of *LNCS*. Springer, 2005.
[34] S. Cruanes, G. Hamon, S. Owre, and N. Shankar. Tool integration with the evidential tool bus. In *VMCAI*, volume 7737 of *LNCS*, pages 275–294. Springer, 2013.
[35] Mario Gleirscher, Simon Foster, and Yakoub Nemouchi. Evolution of formal model-based assurance cases for autonomous robots. In *Software Engineering and Formal Methods: 17th International Conference, SEFM 2019, Oslo, Norway, September 18–20, 2019, Proceedings 17*, pages 87–104. Springer, 2019.
[36] G. Aiello, R. D. Bernardo, M. Maggio, D. D. Bona, and G. L. Re. Inferring business rules from natural language expressions. In *2014 IEEE 7th International Conference on Service-Oriented Computing and Applications*, pages 131–136, 2014.

Ran Wei is a Research Assistant Professor in the Department of Engineering, University of Cambridge. His research interests include Model Based System Engineering, Model-Based Digital Twin Systems, System Design using Twin Systems, High Integrity Systems Engineering and Model Based System Assurance. He is a contributing member of the GSN (Goal Structuring Notation) standard and the SACM (Structured Assurance Case Metamodel) standard. He can be reached at: rw741@cam.ac.uk.



Zhe Jiang received his Ph.D. from University of York (2019). He is currently working as a research associate in University of Cambridge. He is research interests include safety-critical system, system architecture, and system micro-architecture. He can be reached at: zj266@cam.ac.uk.

Haitao Mei received his Ph.D. degree from the Real-Time Systems Research Group at the University of York in 2018. His research interests are in real-time operating systems and programming languages, Big Data and real-time stream processing.

Konstantinos Barmpis is an Assistant Professor in the Auotmated Software Engineering research group in the Department of Computer Science at the University of York. His expertise include MDE (Model-Driven Engineering), DSLs (Domain-Specific Languages) and Repository Mining.

Simon Foster is an Assistant Professor in Computer Science at the University of York. His interests lie in theorem proving, formal semantics, cyber-physical systems, hybrid systems, process algebra, denotational semantics, algebraic methods, and functional programming.

Tim Kelly is a professor of the High-Integrity Systems Engineering research group of the University of York. His research interests include system assurance cases, safety-critical systems engineering, and model based system assurance. He was one of the founding members of the GSN (Goal Structuring Notation) standard and is one of the leading contributors of SACM (Structured Assurance Case Metamodel).

Yan Zhuang is a Professor in School of Artificial Intelligence at Dalian University of Technology, China, leading the Intelligent Robotics Lab (DUT Robotics Lab). His research interests include intelligent sensing, modelling, scene recognition and understanding for mobile robots and other autonomous systems.