

# Concept of Automated Testing of Interactions with a Domain-Specific Modeling Framework with a Combination of Class and Syntax Diagrams

Vanessa Tietz<sup>a</sup> and Bjoern Annighoefer<sup>b</sup>

*Institute of Aircraft Systems, University of Stuttgart, Pfaffenwaldring 27, Stuttgart, Germany*


**Keywords:** Domain-Specific Modeling, Safety-Critical, Avionics, Certifiability, Testability.


**Abstract:** Domain-specific modeling (DSM) is a powerful approach for efficient system and software development. However, its use in safety-critical avionics is still limited due to the rigorous software and system safety requirements. Regardless of whether DSM is used as a development tool or directly in flight software, the software developer must ensure that no unexpected misbehavior occurs. This has to be proven by defined certification processes. For this reason, DOMAINES, a DSM framework specifically adapted to the needs of safety-critical (avionics) systems, is currently being developed. While it is possible to create and process domain-specific languages and models, the challenge lies in ensuring that the framework consistently performs as intended, providing the foundation for certification. For this purpose, a novel approach is employed: the introduction of a meta-meta-modeling language that combines syntax diagrams with a class diagram. This language serves as a comprehensive reference for the generation of test cases and the formal linking of grammar, meta-modeling language and implementation. This allows the implementation to be tested with every conceivable command. In addition, mechanisms ensure that this set of commands to be tested is a closed set.

## 1 INTRODUCTION

Domain-specific modeling (DSM) enables the efficient development and design of systems as well as software and is to be considered as state-of-the-art in many engineering domains. Avionics is a domain that has only relied on DSM methods to a small extent. Here, DSM is mainly used to assist development processes. The complete development of avionics systems often fails because DSM tools either have to be qualified or the output generated from these tools has to be verified and validated in a time-consuming way. In our previous study (Tietz et al., a), the reasons for this are analyzed and suggestions are made as to what a DSM tool would have to look like. Within our own DSM tool DOMAINES (presented in (Tietz et al., b)), it is possible to create and edit domain-specific languages and domain-specific models with simple command line inputs. The proof that this implementation can be used in the safety-critical avionics domain has not yet been provided and is the subject of this paper. To be allowed to use a tool in a safety-critical environment such as avionics, it must

be proven that this tool behaves correctly and that its output cannot lead to unexpected misbehavior in flight operations. For this purpose, evidence must be provided as defined in standards and norms. The exact manner in which this evidence is provided is left to the discretion of the developer. One conceivable way to provide this evidence is to confront the framework with every conceivable input and verify that it behaves as intended. At first glance, the set of all conceivable commands seems infinite. However, by selecting suitable algorithms, a filtering effect can be achieved that reduces the number of commands to be considered and thus enables this approach. The most effective way is to implement a parser that can determine, whether a command can proceed or not. This requires a suitable grammar that contains information about the implemented meta-modeling language. To achieve this, a meta-meta modeling language is proposed which combines syntax diagrams with a class diagram. This new language can serve as a single-source-of-truth for deriving test cases, and to establish a formal connection between grammar, model and implementation. One advantage of this approach is that the grammar and the test cases can be automatically adapted if the modeling language is changed.

<sup>a</sup>  <https://orcid.org/0000-0002-5942-5893>

<sup>b</sup>  <https://orcid.org/0000-0002-1268-0862>

## 2 FUNDAMENTALS

### 2.1 Avionics Software Development

Avionics software always operates within a larger system. Aviation standard ARP 4754A (SAE, 2010) outlines suitable methods for demonstrating compliance with airworthiness regulations during aircraft system development. The process model in ARP 4754A is a V-model. The software life cycle is divided into a development process and a verification process, which are represented by the left and right branches of the V-model, respectively. In the verification process, it must be tested that the implementation meets defined requirements. The software's influence on aircraft safety and criticality determines its assigned software development assurance level (DAL). The software level outlines the objectives and activities of the software life cycle. At the foundational level of the avionics software development process resides the source code. As per DO-178C (referenced by the ARP 4754A), the source code for DAL C software must possess the traits of being traceable, verifiable, consistent, and correctly implementing low-level requirements (RTCA, 2011).

### 2.2 Domain-Specific Modeling Wordings

Due to differing interpretations of terms and levels within the DSM field, a brief explanation of the terminology used will be provided here:

- M0: A real world application.
- M1 - **User model**: Digital representation of a real world application.
- M2: **Domain-specific language**: Blueprint of a specific application enabling the modeling of a domain-specific model.
- M3: **Meta-modeling language**: Language enabling the modeling of a domain-specific language. Typically part of a software framework.
- M4: **Meta-meta-language**: Language of the meta-language.

## 3 DOMAINES

With **DOMAINES (Domain-Specific Modeling for Aircraft and other Environments)**, we are developing a domain-specific modeling framework that is specifically designed for the use in the avionics domain. It is intended to be used in the systems development, as well as in flying software. For the purposes of this paper, only the core of **DOMAINES** remains of interest.

### 3.1 The DOMAINES Core

The basis of the **DOMAINES Core** is the simplified meta **MODELing Language (MOD)** operating on M3 level, which is implemented with the programming language Ada in the **RUNtime ModelDataBase (RUNMDB)**, as depicted in Figure 1. The model interaction, i.e., the creation as well as the editing of domain-specific languages and user models is conducted through the interface with the **Basic Model Interaction (BMI)** interface. The BMI is part of a generic interface following the **Essential Object Query (EOQ)** formalism (Annighoefer et al., 2021) which is a model query language. The purpose of this interface is to translate complex model requests and model operations (queries) received from other peripheral technology into BMI suitable commands. Peripheral technology could be for example a model transformation engine or a graphical editor.

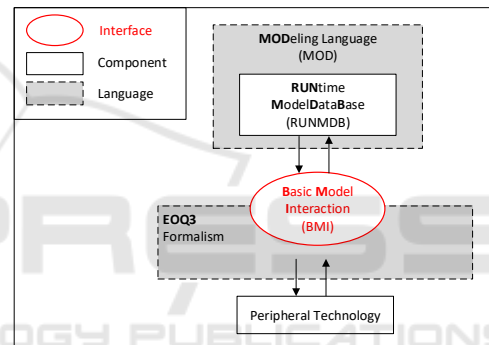


Figure 1: DOMAINES Core.

Our initial step is to ensure that **RUNMDB** is certifiable. For this, the interface between **BMI** and **RUNMDB** is crucial. Everything that happens from this interface onwards must be free of unexpected misbehavior. This proof must be provided to authorities. This also means that the interface as well as the **RUNMDB** itself must be designed in such a way that the required evidence can be provided as easily as possible.

### 3.2 Runtime Model Database (RUNMDB)

The **RUNMDB** is basically the implementation of **MOD** and thus allows to create, edit, read, and delete elements of models on M2 and M1 level. The structure of the **RUNMDB** is depicted in Figure 2.

The elements shown in green are external data needed for command processing. A command sent from **BMI** to **RUNMDB** is first processed by a *Parser*, which decides whether a command is valid or not ac-

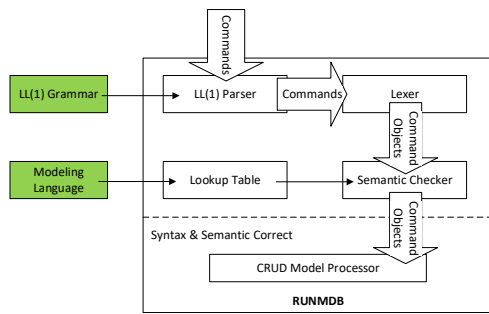


Figure 2: Overview RUNMDB.

According to a given *Grammar*. The syntactically correct commands are then decomposed and processed into command objects in the *Lexer*. Afterwards, these commands are checked for semantic correctness using *Meta-Modeling Language* information in the *Semantic Checker*. Subsequently, the expected set of commands processed in the *ModelProcessor* will only comprise semantically and syntactically correct commands assuming that the *Parser* and *Semantic Checker* behave correctly. The *ModelProcessor* is solely responsible for processing the commands. This is the usual functionality that is also covered by other modeling frameworks such as EMF, GME, or the Mathworks System Composer. The unique selling point of our approach is the certifiability and thus the unconditional applicability in the area of safety-critical (avionics) systems.

## 4 ON REACHING CERTIFIABILITY OF RUNMDB

Software certification requires demonstration of its intended functionality and absence of unexpected errors during flight. Since the primary goal is to know and correctly catch any possible misbehavior, the idea is to test the behavior of every conceivable command within the different parts of the RUNMDB. At first glance, this seems an infinite number of commands, and therefore, impossible. However, by choosing appropriate mechanisms and definitions, a filtering effect can be achieved, which can reduce the number of commands to be considered. This turns an infinite set of commands to be considered into a significantly reduced manageable set. In the following, requirements are formulated for the components in terms of their testability.

### 4.1 BMI-RUNMDB Interface

The BMI-RUNMDB interface is responsible for transmitting commands from the peripherals. Fur-

thermore, the interface defines the pattern of the commands to be expected. It must be designed in such a way that the commands

- can be formally defined in order to ease testability.
- have a clear and decomposable structure.
- can be represented using a closed set.

### 4.1.1 CRUD Commands

Based on the requirements, the decision was made to utilize text-based CRUD (**C**reate, **R**ead, **U**psert, and **D**elate) commands. They can be formalized through the use of an (Extended) Backus-Naur Form ((E)BNF). Every CRUD command always has a *Target* as first element. The *Target* indicates which element within a model the command refers to. The *Read* and *Update* commands additionally have a *Feature*, specifying the property of an element. To be able to set values with an *Update* command a *Value* and its *Position* within a possible list is needed (by default 1). Since the *Features* are defined within the *Meta-Modeling Language*, this is a closed set of possibilities. For *Target* and *Position*, the closed set is achieved by the implementation property of static lists. Only in the case of *Value* can a closed set not be assumed across the board. For this, suitable arguments must be found in order to be able to justify that every conceivable command can be tested.

### 4.2 Parser

The *Parser's* objective is to determine the validity of an input command. The parser must be designed in such a way that

- every legal command can be tested.
- every incomplete command can be tested.
- every illegal command can be tested.

When testing it has to be ensured that

- a legal command, is recognized as valid.
- an incomplete command, is recognized as invalid.
- a illegal command, is recognized as invalid.

To meet the requirements, an LL(1) parser was chosen. This is a top-down parser with a look-ahead of a single-character. Meaning that the decision whether a command is valid or not depends only on the next character.

### 4.3 Lexer

The *Lexer* decomposes a syntactically correct command into objects suitable for further processing. For

the decomposition of the commands it is important that this proceeds deterministically in order to be able to generate the original commands from decomposed objects when testing the behavior.

#### 4.4 Semantic Checker

While the *Parser* is responsible for checking the syntax of a command, the *Semantic Checker* is responsible for checking the correct semantics. The semantic checks include but are not limit to the checking so that

1. every element addressed is available
2. addressed features are available
3. only editable features can be updated
4. every part operates on the same level
5. feature and value match
6. target and features match
7. that limitations within the implementation are correctly taken into account

For testing the correct behavior of every semantic check it has to be ensured that

- the test cases are automatically derivable
- a correlation between the *Grammar* and the *Meta-Modeling Language* exists to enable the possibility to either have a more complex grammar and less semantic checks or vice versa.

The goal is to eliminate invalid commands as much as possible so that as little effort as possible is required in the *ModelProcessor* to catch invalid behavior.

#### 4.5 ModelProcessor

The *ModelProcessor* processes syntactically and semantically correct commands. It performs four different model operations, based on the CRUD command structure: creating elements, updating and reading model information, and deleting elements. The operations within the *ModelProcessor* are time-dependent. As a result, an infinite number of states must be checked. Thus, the *ModelProcessor* must be designed so that

- the complete behavior can be tested.
- fixed sized lists should be used.

#### 4.6 Problem Statement

As seen in Figure 2, the *Grammar* and the *Meta-Modeling Language* have no origin or relationship to

each other. Since the requirements for testing described heavily rely on the *Grammar* and the *Meta-Modeling Language*, a claim about the correct behavior of the RUNMDB can only be formulated once a formal correlation between the (implemented) *Meta-Modeling Language* and the *Grammar* has been established.

The *Meta-Modeling Language* can be considered as a requirements document for the implementation of RUNMDB. The *Grammar* serves as input to the parser and thus forms the basis for deciding whether a command may be processed further or not. The more restrictive the grammar, the fewer semantic checks are necessary, resulting in less need for testing procedures. More restrictiveness within the grammar can be achieved by including information from the *Meta-Modeling Language*. Furthermore, in the area of testing, it is useful to have a single source of truth from which tests can be automatically derived. A combination of *Grammar* and *Meta-Modeling Language* could represent such a single source of truth. At least for testing the *LL(1) Parser*, the *Lexer* and the *Semantic Checker*, the test cases can be derived from the single source of truth, since there are no temporal dependencies between the commands. This allows us to formulate the following research question:

**Is there a way to combine a grammar with a modeling language to enable ...**

- the creation of a strongly restrictive *LL(1) Grammar*?
- the introduction of a single source of truth?
- a formal correlation between *LL(1) Grammar* and *Meta Modeling Language*?

## 5 ESTABLISHING A RELATIONSHIP BETWEEN GRAMMAR AND MODEL

Section 4.6 identified the necessity for a formal correlation of LL(1) grammar and meta-modeling language to assist the certification process. Since a context-free grammar can also be represented graphically with syntax diagrams, and modeling languages are inherently graphical, the idea of combining the graphical notation of a grammar with a typical class diagram is a natural idea.

### 5.1 Syntax Diagrams

A context-free grammar can be represented by several syntax diagrams (Braz, 1990). A syntax diagram rep-



resents a single production rule of a grammar. An entry point can be connected via various paths with terminal and non-terminal symbols to an end point. Terminal symbols are represented by circular boxes and non-terminal symbols by rectangular boxes. Repetitions of non-terminal symbols are denoted by curved lines. An example of the representation of a context-free grammar for the definition of the *Create* command with syntax diagrams is depicted in Figure 3.

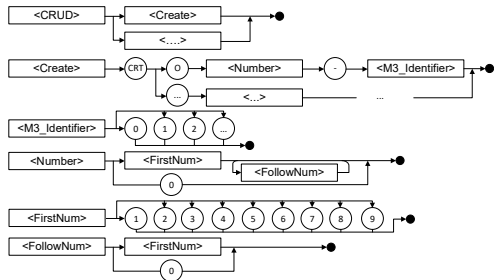


Figure 3: Example of syntax diagrams.

The non-terminal `<CRUD>` can be created through the non-terminal `<Create>`, or other non-terminals. The non-terminal `<Create>` can consist of the terminals "CRT", and "O" and the non-terminal `<Number>` followed by the terminal "." and the non-terminal `<M3Identifier>`. All other syntax diagrams follow the same structure and meaning.

### 5.2 On Combining Class and Syntax Diagrams

Within the DOMAINES framework the implemented meta-modeling language operates on M3 level to enable the interaction with models on M2 and M1 level. The concept of combining a grammar and a meta-modeling language is essentially to create a common template for the grammar and the meta-modeling language. In the modeling environment, this is usually achieved by going one meta-level deeper.

To integrate syntax diagrams with a class diagram, non-terminals are substituted with class elements, virtually as a placeholder for information that is filled by the design of the overlying M3 language. To maintain their recognizability as placeholders within the text-based grammar, these classes are denoted by a + preceding their name. In Figure 4 the combination of a syntax diagram with a class diagram is depicted for the *Create* command.

This syntax diagram describes the notation needed to create a model (M) (either on M2 or M1 level), an element at M2 level (O) or an element at M1 level (I). The class diagram describes the structure of the M3 language. A class on M3 level is always of type

*StructureElement* and contains a `<+M3Identifier>`, which is needed to create respective elements on M2 level. The `<+M3Identifier>` is present in both the syntax diagram and the class diagram and serves as a placeholder for the textual representation. It is filled as soon as an M3 language is instantiated from this M4 language. Each *StructureElement* has features (attributes) whose structure can also be seen in the class diagram.

### 5.3 Processing

The entire process from an *M4 Language* to the final *LL(1) Grammar* is depicted in Figure 5. The green boxes represent textual documents, while the gray ones depict graphical representations.

Having developed the *M4 Language* it is possible to manually draw an *Initial Grammar* based on the syntax diagrams. Within that *Initial Grammar*, there are still non-terminals that serve as placeholders and are marked with a + at the beginning. The grammar is created by following the path from the first non-terminal to the end symbol of the syntax diagram. When branches are reached, additional non-terminals are created in the grammar, named `<Branch>` and supplemented with an incremented number. An example of an *Initial Grammar* based on the *M4 Language* in Figure 4, with the placeholders in red, is listed below:

```

<CMDCreate> ::= "CRT" <Branch1>
<Branch1> ::= "M" <Branch2> | "O" <Number> "-"
<+M3Identifier> | "I" <Number> "." <Number>
"-" <+M3Identifier> "." <Number>
<Branch2> ::= <Number> <Branch3> | "x"
<Branch3> ::= "-x"
<Number> ::= <FirstNum> <FollowNum>* | "0"
<FirstNum> ::= "1" | ... | "9"
<FollowNum> ::= <FirstNum> | "0"
    
```

With the class diagram as part of the *M4 Language* it is possible to draw the implemented *M3 Language*. The graphical representation of the meta-modeling language is only used for the descriptive demonstration of the functionality. To combine language with grammar, a text-based representation is needed. As text format for the modeling language the json file format was selected, because this format is easily readable by human and is efficiently processed by computers. The text-based *M3 Language* includes counterparts of the placeholders in the *Initial Grammar*. A snippet of the text-based modeling language is listed below.

```

"Class" : {
  "<+M3Identifier>" : {
    "name" : "1",
    "id" : 1
  }
}
    
```

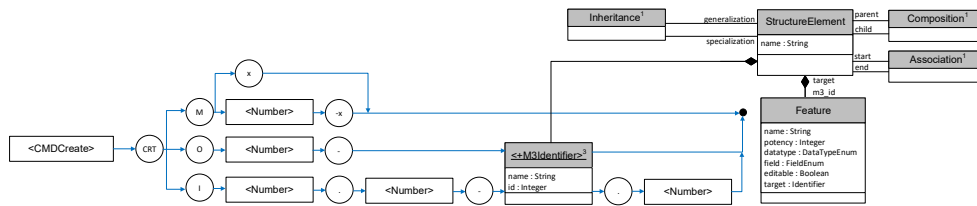


Figure 4: Example of a combination of syntax diagrams with a class diagram for the create command.

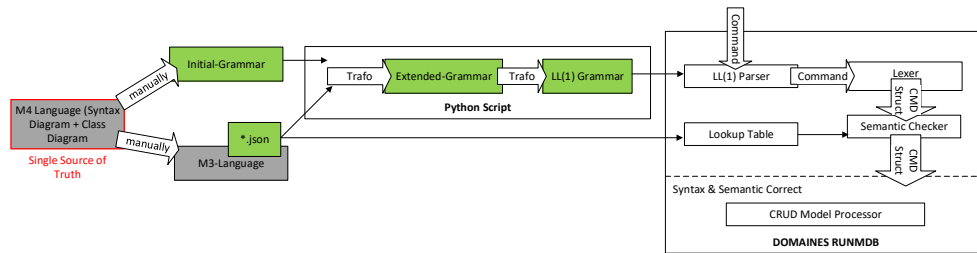


Figure 5: From M4 Language to a Parsable LL(1) Grammar.

```

    },
    "<+M2M1Feature>" : {
        "name" : "clabjectid",
        "potency" : 2,
        "field" : "Dual",
        "datatype" : "Integer",
        "editable" : "False",
        "m3_id" : 1,
        "target" : 1
    }, ...
    
```

This is where the placeholder `<+M3Identifier>` is located again, the information contained there can be taken over thereby into the *Initial Grammar*. The information inside the placeholder is always stored in the *name*. Accordingly, the “name” is searched for within the feature and this information is stored in the grammar. After having transferred the information about the `<+M3Identifier>` to the *Initial Grammar* an *Extended Grammar* as an intermediate step is available. This extended grammar looks as follows:

```

<CMDCreate> ::= "CRT" <Branch1>
<Branch1> ::= "M" <Branch2> | "0" <Number> "-"
<+M3Identifier> | "I" <Number> "." <Number>
 "-" <+M3Identifier> "." <Number>
<Branch2> ::= <Number> <Branch3> | "x"
<Branch3> ::= "-x"
<+M3Identifier> ::= "1"
<Number> ::= <FirstNum> <FollowNum>* | "0"
<FirstNum> ::= "1" | ... | "9"
<FollowNum> ::= <FirstNum> | "0"
    
```

It can be seen that another line has been added with the information from the *M3 Language*. However, parsability with an LL(1) parser is not yet guaranteed.

## 5.4 Effects on Certifiability

Demonstrating that the intended functionality of the framework has been implemented is a requirement of certification authorities. The intended functionality is described by the *M3 Language*. Via the relationship between the *LL(1) Grammar* and the *M3 Language*, the complete intended functionality of the implementation can be systematically tested.

### • LL(1) Parser:

- **Valid commands:** The first step is to generate a syntax tree based on the *LL(1) Grammar*, which can be used to navigate through a grammar and therefore to generate all valid commands. The fact that this is a closed set has already been shown in 4.2, only for the *Value* of a command an explanation is still missing. We refer to the property of the LL(1) parser, where only the next character is considered and based on that it is decided if the command is valid or not. Thus the length and the permutation possibilities available with it are irrelevant, since the parser behaves identically and thus only each possible character within a value must be tested once. This is based on the assumption that a closed set of characters is used to describe a value.
- **Incomplete commands:** For this purpose, all valid commands are used and broken down into individual characters.
- **Invalid commands:** For this, the property of the *LL(1) parser* with the look-ahead of a single character is utilized. No matter which position within the grammar is currently being con-

sidered, it is possible to find all possible following characters. From the complementary set it is possible to generate every non-valid command. Having a look at the *LL(1) Grammar* from 5.3, the first line indicates that the first possible character is a "C", all other characters are invalid. Hence, commands with all invalid characters have to be recognized as invalid. In the next step starting with the valid character "C". Based on that the next valid characters ("R") are determined. Every other character is invalid and is appended to the "C" and send to the *LL(1) Parser*. For example, this causes the commands "CA", "Ca", "CB", "Cb", ... to be invalid.

With that the other components of RUNMDB need only be tested with an immensely smaller set of commands. The *M3 Language* information contained in the *LL(1) Grammar* eliminates the need for some semantic checks. For instance, the first four semantic checks from Section 4.4 are already caught in the grammar since, e.g. only existing  $\langle +M3Identifier \rangle$  can be created or all possible features are already stored in the *LL(1) Grammar*. By increasing the complexity of the grammar, more semantic properties can be introduced, but this leads to the fact that the checking of a command for validity takes longer and thus a trade-off between the runtime and the less required semantic checks must be made.

- **Lexer:** Only all syntactically correct commands must be considered. The command objects created in the lexer are put together into strings and checked whether these strings correspond to the original command.
- **Semantic Checker:** For the remaining semantic checks, test cases need to be generated. These can be mostly automatically derived from the *M3 Language* and information about list sizes from the implementation and follows white box testing approaches (IEEE, 2021). For example the eighth semantic check in Section 4.4 examines if the program operates correctly when list sizes are exceeded. This may occur if the selected *Position* is outside the defined list size. Therefore, all possible update commands are generated from the grammar and the *Position* parameter is varied beyond the list limits. Having a *Position* which is out of bounds, an error message has to be generated. All further test cases for the semantic checks are generated in a similar way. The test cases collectively result in a manageable closed set of commands. By basing test cases on a closed set of

valid commands or the meta-modeling language, the testing of semantic checks can be fully traced back to a finite set of test cases.

- **CRUD ModelProcessor:** The *ModelProcessor* operations are time-dependent on each other, resulting in an infinite number of states to be checked. The implementation is based on the CRUDs, so there is a separate operation for **Create**, **Read**, **Update**, and **Delete**. Within each CRUD, the set to be tested can be reduced to the following scenarios. It is notable that these scenarios rely on static lists in their implementation and the first-in, first-out (fifo) principle.
  - **Create:** Elements are created and stored in static lists depending on their type and remain on their position until they are deleted.
    1. Create an element at the end of a list
    2. Create an element between existing elements
    3. Try to create an element at the end of a full list
  - **Read:** The values can be either in fixed sized lists or in single entries. Read
    1. information from an empty model
    2. updated information from single values
    3. updated information from list values
    4. information from full lists
  - **Update:** If values are stored in lists, the position can also be specified.
    1. Update every feature with a value
    2. Update with a value at every possible position
    3. Try to add values to fill lists
    4. Change references
  - **Delete:** All values of the deleted object are reset to their default values. Delete elements
    1. where no information was updated
    2. with updated values and full value lists
    3. with updated references

Any behavior, at any time, can be traced back to one of these scenarios. This procedure enables the limitless set of combinations to be narrowed down to a finite set of scenarios that can be tested. These explanations have shown that it is feasible to test the implementation for any possible command by utilizing a combination of a *Meta-modeling language* and a *LL(1) Grammar*, in order to avoid any unexpected behavior. This provides an important foundation for a potential certification process for RUNMDB. At present, at least the tests for the *LL(1) Parser*, the *Lexer*, and the *Semantic Checker* can be generated automatically from the single source of truth available through the combination of *LL(1) Grammar* and *Meta-modeling*

language. For the *ModelProcessor*, there is currently no automated solution for generating test scenarios and their evaluation.

## 6 RELATED WORK

With our approach we strive for a certifiable framework for DSM in order to fully exploit the advantages of DSM in the safety-critical (avionics) environment. We are not the first to do so. For example, Matlab Simulink claims to have a Tool Qualification Kit for the DO-178C standard. In the field of certifiable code generators are additional approaches such as the most prominent example SCADE (Dormoy, 2008). Another one is the Gene-Auto code-generator (Toom et al., 2008) or TargetLink from dSpace (dSpace, ). Each approach mentioned implies that certification or qualification is attainable. However, the process of achieving this remains undisclosed. There is no method published against which we may compare our approach.

The idea of how we want to achieve certifiability is not entirely new. The basis is testing the implementation through some kind of grammar-based testing which is subject to research in (Sirer and Bershad, ), (Sharma, ). This involves deriving test cases from a formally defined grammar to systematically test a system or software with its input. One common issue with that approach is the possibility of generating an infinite number of test cases which is addressed in (Hoffman et al., 2011). An infinite number of test cases can result in test coverage reaching its limits, this is addressed in (Godefroid et al., 2008) via fuzzing. Our approach solves the problem of infinite test cases by combining our meta-meta-modeling language with the grammar on the one hand, and using an LL(1) parser that can guarantee a closed set of test cases on the other hand.

Beside that, we are unaware of any other correlated research.

## 7 CONCLUSION AND OUTLOOK

In this paper, we presented a new M4 meta-meta modeling language, which is required in our highly safety-critical avionics domain to be able to prove that the implementation of the meta-modeling language can be certified. This is based on the fact that every conceivable command in the implementation can be tested to show that no unwanted misbehavior occurs. In order to make this possible at all, a formal definition of commands within a grammar and

a corresponding parser are used. A statement about the functionality of the implementation is only possible, if a formal correlation between grammar and implemented meta-modeling language exists, which is achieved by the proposed M4 meta-meta modeling language. In addition, the M4 meta-meta modeling language acts as a single source of truth. This ultimately should make it suitable for the development of safety-critical avionics as well as for use in airborne software. For further work it is important to assess whether the current test cases adequately test all statements within the implementation. Ideally, statement coverage should adhere to the common avionics standards outlined in DO-178C.

## ACKNOWLEDGEMENTS

The German Federal Ministry for Economic Affairs and Climate Action (BMWK) has funded this research within the LUFO-VI program.

## REFERENCES

- Annighoefer, B., Brunner, M., Luettig, B., and Schoepf, J. (2021). EOQ: An Open Source Interface for a More DAMMMN Domain-specific Model Utilization. In *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*.
- Braz, L. M. (1990). Visual syntax diagrams for programming language statements. *ACM SIGDOC Asterisk Journal of Computer Documentation*, 14(4):23–27.
- Dormoy, F.-X. (2008). SCADE 6 a model based solution for safety critical software development. In *Embedded Real Time Software and Systems (ERTS2008)*.
- dSpace. Targetlink. [https://www.dspace.com/de/gmb/home/products/sw/pcgs/targetlink.cfm#176\\\_25806](https://www.dspace.com/de/gmb/home/products/sw/pcgs/targetlink.cfm#176\_25806). Accessed: 2023-09-20.
- Godefroid, P., Kiezun, A., and Levin, M. Y. (2008). Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation*, pages 206–215.
- Hoffman, D. M., Ly-Gagnon, D., Strooper, P., and Wang, H.-Y. (2011). Grammar-based test generation with YouGen. *Software: Practice and Experience*, 41(4).
- IEEE (2021). ISO/IEC/IEEE International Standard - Software and systems engineering - Software testing – Part 2: Test processes. Standard, IEEE.
- RTCA (2011). DO-178C software considerations in airborne systems and equipment. Standard, RTCA.
- SAE (2010). Guidelines for development of civil aircraft and systems. Standard, SAE.
- Sharma, A. How does grammar-based test case generation work? <https://www.veracode.com/blog/managing-a-pptest/how-does-grammar-based-test-case-generation-work>. Accessed: 2023-10-04.



- Sirer, E. G. and Bershad, B. N. Using production grammars in software testing. *ACM SIGPLAN Notices*, 35(1).
- Tietz, V., Frey, C., Schoepf, J., and Annighoefer, B. Why the use of domain-specific modeling in airworthy software requires new methods and how these might look like? In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pages 627–632.
- Tietz, V., Schoepf, J., Waldvogel, A., and Annighoefer, B. A concept for a qualifiable (meta)-modeling framework deployable in systems and tools of safety-critical and cyber-physical environments. In *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*.
- Toom, A., Naks, T., Pantel, M., Gandriau, M., and Wati, I. (2008). Gene-auto: an automatic code generator for a safe subset of simulink/stateflow and scicos. In *Embedded Real Time Software and Systems (ERTS2008)*.

