Iris Groher, Thomas Vogel (Hrsg.): Software Engineering 2023 Workshops,
92 Digital Library, Gesellschaft für Informatik, 2023

A Domain-Specific Language for Requirements Engineering in Safety-Critical Automotive Software Development

Stefan Schlichthaerle[1] Philipp Wullstein-Kammler[2] Florian Schanda[3]

Abstract: Requirements engineering is a crucial part of every software development project. Deficits in this discipline have tremendous impact on the overall project's success.

In this paper, we present our approach that treats requirements like source code and therefore benefits from modern software development workflows and paradigms, to bridge the gap between requirements engineering and large-scale agile software development. We derive the requirements for our approach and discuss the underlying tooling as well as the corresponding impact on processes. We show examples, including validation of requirements and the traceability towards source code.

Keywords: Requirements engineering, Agile software development, Domain-specific language

1 Introduction

Innovation in the automotive domain is more and more driven by software [Fü10, Br06]. State of the art vehicles released in 2020 contain more than 100 million lines of code, distributed among dozens electronic control units (ECUs), that are developed by multiple parties and are eventually integrated into one complex technical device [Mi21]. Requirements engineering is not a new discipline. In the automotive domain, it has been used for decades to handle complex supplier relationships and to describe all the pieces that form a vehicle, regardless if they are physical devices or software products [Al09]. An agile software development process, as well as the ever increasing complexity of software, demands an evolution of the established processes, methods, and tools.

There are three obvious ways you can write requirements and code:

- Separate: code and requirements are separate but linked artefacts.
- In code: code is the main artefact, requirements are special comments. Examples of this include Doxygen or Sphinx.
- In requirements: the requirement is the main artefact, code is embedded in it. This is called literate programming. [Kn86]

[1] BMW Group, 80788 Munich, Germany, stefan.schlichthaerle@bmw.de
[2] BMW Group, 80788 Munich, Germany, philipp.wullstein-kammler@bmw.de
[3] BMW Group, 80788 Munich, Germany, florian.schanda@bmw.de

doi:10.18420/se2023-ws-11

At BMW, requirements engineering follows the *separate* pattern, with the additional caveat that the requirements are tied to databases totally detached from the source code, as it is common with the usual requirements engineering tools such as DOORS, codebeamer, SystemWeaver, etc. A specific pain point of these approaches is that code and requirements need to be kept in sync; and occasionally variants (i.e. branches) need to be created, maintained and possibly re-integrated. All tools – that we know of – operating on a separate detached artefact make this process painful and costly: several full-time positions are used just to deal with this, and provide internal extensions to these tools to make them more usable.

We decided to improve the existing process by bringing the requirements closer to the code. Our approach is called *Treat Requirements Like Code*. We represent all requirements and their metadata in a domain-specific language and place them into the source repository, so that all development processes that apply for source code, tests, as well as architectural models [SBS20] can be applied for the requirements as well. Note that the domain here is *requirements engineering*, the language is *not* specific to the automotive domain at all.

This solves both pain points identified above: we no longer need to perform a manual activity to keep or demonstrate code/requirements correspondence (they always are); and we can use modern version control systems like git to do branching, merging, diffing and reviewing. In addition checking requirements and their relationship to code in CI allows us to maintain any argumentation we need as we develop, branch, and merge the code base.

In this paper, we examine this approach in a large agile software development project at BMW, which focuses on developing the software platform for driver assistance and automated driving. We also introduce the domain-specific language as well as the environment and processes, that are necessary to enable this approach.

2 Requirements and Goals for Requirements Engineering

According to ASPICE SWE.1, software requirements analysis is the basis to define scope and functionality of the software product. To properly implement it, traceability between requirements and the corresponding source code and test artefacts has to be established [VD17]. ISO 26262 [IS18], the norm that defines safety-related automotive software development, defines the specification of software safety requirements as mandatory phase of the product development at the software level and requires a mapping from technical safety requirements towards software. Therefore a key requirement for our requirements engineering processes and tooling is **compliance with ASPICE and ISO 26262**, so that the defined approach can be clearly linked into the respective norms and process frameworks.

Agile software development is increasingly popular in all areas of software development, including embedded software for cyber-physical systems [Sh12]. Its lean structure is focusing on the software product itself, so that there is always a shippable product available. The

ability to adapt to changed circumstances, or requirements, is a fundamental aspect of agile software development. Therefore another requirement for our requirements engineering processes and tools is **compatibility with agile software development**, which means it shall strengthen the agile approach and not contradict it for example by introducing tool breaks or heavyweight processes, which prevent fast iteration cycles.

Fast iteration cycles are a key element of agile software development [Be01]. They enable fast adaption to changed requirements, for example the latest findings from customer surveys, as well as previously unknown technical challenges in the technology stack used. To support this approach, it is important to have **all artefacts** of the software product, namely requirements, source code, tests on all levels, and documentation, **in a consistent, linked representation**, so that the actual change can be applied consistently to all artefacts. Specifically, using database tools and trying to integrate them into the agile development process, often leads to fundamental clashes and incompatibilities, resulting in asynchronous change processes with different timelines, that prevent consistent artefacts [SBS20].

In order to strengthen the aspect of a consistent change, that can be applied to all artefacts of the software product, a harmonised workflow aligned to changes in the source code is another requirement for our requirements engineering processes and tools. Managing source code in a repository and applying changes via a pull request is the de facto industry standard for large software projects. Thus we aligned to this approach, and decided to manage requirements with text-based domain-specific language directly in the repository, right next to the source code that has to fulfil the requirements. Another aspect is that, in our experience, the acceptance of software developers towards requirements engineering increases since they can **apply their existing workflows and tools** to this kind of engineering work as well.

3 Related Work

In [LB15], the authors describe the usage of a domain-specific language for requirements engineering in the context of IT development. In a concrete project, they use a DSL to describe customer requirements as well as the resulting use cases that shall be fulfilled by the IT system. One key benefit they identified is the fact that the DSL can be formally processed and validated, is also accessible and understandable to non-IT users of the resulting product, and thus fosters the overall requirements elicitation process. They also emphasise benefits on report generation and well as artefact transformation.

The author of [Ga18] is proposing a domain-specific language for requirements engineering, to reduce inconsistency between requirements and resulting implementation and test. The Goal of the DSL is to address all stakeholders, so that formal specifications can also be handled by developers not familiar with them. The paper proposes to use an existing programming language, so that the requirements can be executed, similar to a test case. In addition, the language is extended with linkage abilities, so that the requirement can link into source code as well as other documents.

4 Treat Requirements Like Code (TRLC)

TRLC is an open[4] framework that enables requirements engineering based on source files located in a repository. The TRLC language itself, which is part of the framework, is a simple but extensible text-based language to express requirements written in natural or formal language, and linking to other items. In addition to linking, which is essential for software traceability, in large organisations there are often rules on meta-data. For example "an ASIL-D requirement must be traced to a safety goal".

We should be clear about the scope: TRLC is a carrier format, it is *not* in itself a method to write requirements. It neither encourages or discourages any approach. You could write your requirements in a natural language; or you could write them in Z[IS02]. An example of some formality is the TRLC LRM[5] (Language Reference Manual) itself, where we intend to validate the BNF grammar contained within, the language examples quoted, and generate parts of its own lexer from itself. An example of some semi-formality in the automotive domain could be generating configuration data from TRLC requirements.

The TRLC language consists has three major parts: type definitions, check definitions, and requirements declarations. The type definitions introduce enumerations and records, which can later be used to write requirements. There is no built-in requirement type, as one of the design goals of TRLC is to not have any built-in business logic. You will need to make one that is suitable for your needs. For example:

```
enum ASIL {QM A B C D}

type Requirement {
  description String
}

type Safey_Requirement extends Requirement {
  asil                  ASIL
  derives_from  optional Requirement
  codebeamer_id optional Integer
}
```

The check definitions add custom constraints for validating requirements:

```
checks Requirement {
  len(description) < 10,
    warning "isn't this is a bit too short?"
  }
}

```

[4] TRLC and its tools are free software: `https://github.com/bmw-software-engineering`.
TRLC (the language) is released under the GFDL1.3,
TRLC (the tools) are released under the GPL3, and
LOBSTER (the traceability tools) are released under the AGPL3.

[5] https://github.com/bmw-software-engineering/trlc/tree/main/language-reference-manual

```
checks Safety_Requirement {
  asil != ASIL.QM implies
    (derives_from != null or
     codebeamer_id != null)
  error "an ASIL requirement must be linked"
}
```

The check language is fairly expressive (it supports Boolean logic, quantifiers, arithmetic, and string manipulation including regular expressions) but it currently has one key limitation: you can only base your check on the current requirement (i.e. you cannot talk about other requirements). We are considering lifting this limitation if there is demand for it, although these kinds of checks can already be implemented with user scripts using the Python API.

Finally, we can use these type to write our requirements:

```
package Example

Safety_Requirement car_integrity {
  description = '''
     We shall not explode, accidentally
     or otherwise.
  '''
  codebeamer_id = 12345
  asil = ASIL.D
}
```

The ease of parsing this format enables a much more fine-grained and accurate software tracing. We can add special comments or pragmas in code, for example:

```
bool should_we_explode () {
  // lobster-trace: Example.car_integrity
  return false;
}
```

These tags can be extracted for any reasonable implementation language or framework and cross-correlated with the LOBSTER[6] tool, which is also published under a free-software license, to produce a software tracing report that is entirely generated from the information stored in your source repository. This can even be done offline, but the main benefit is that you now maintain your tracing evidence *incrementally and atomically on every commit* and you no longer have to worry about it near the delivery date: making a significant improvement and simplification to the agile workflow.

[6] Lightweight Open BMW Software Traceability Evidence Report. In this tool we support TRLC, C, C++, Python, MATLAB or GNU Octave, and GoogleTest. We plan to also support Rust, Java, Kotlin, Ada/SPARK, Simulink, and Franka+, and anything else the community provides.

5 TRLC Modelling Approach

In the following section, we describe in detail how a project setup may look like that uses the TRLC framework as well as the TRLC language for requirements engineering. Starting from the repository structure and corresponding continuous integration infrastructure, we also discuss the key part of configuration as code, which is also enabled by TRLC.

5.1 Repository Structure

As stated earlier, TRLC allows us to put the requirements for a software product into the same repository as the source code.

We recommend to structure the repository in the following way:

- Store the type definitions and their checks in one central location. This helps the requirements management team to establish ownership over the type definitions. Software development teams do not always have an overview of all the requirements management needs of other teams, especially in very large software projects.
- Store the requirements in distributed files as close to the relevant source code as possible, for example in the same directory. This helps the software development teams to find their requirements easily even without proper traceability. The close proximity between source code and requirements can already serve as a simple way of linking requirements to software components or units.
- Use a continuous integration system and install checks that ensure the completeness of requirements before a merge to the main development branch. We will further elaborate on this point in section 5.5.

5.2 Dealing with evolving Software Development Process

Large software development projects often face the problem that the corresponding software development process is evolving in parallel to the project execution. Especially in the beginning of a project, during definition of the initial process and selection of development tools, we cannot reasonably foresee all implications of these choices.

This is generally not a problem for TRLC; as TRLC allows the requirements managers to make decisions as fast as possible and refactor the type definitions later when needed. Changing the structure of requirements retroactively is an activity that is extremely difficult with tools based around external databases, and TRLC makes this activity much easier: part of it can usually be automated and the CI checks can enforce consistency retroactively through the TRLC check language.

A typical evolution of a project using TRLC as its framework for requirements engineering may looks like this:

1. Project start: requirement types and attributes are defined.
2. Early phase: engineers write requirements and source code. The product grows and features are added.
3. Unforeseen event: for example, parts of the software stack are delegated to a collaboration with another company, and it will now be developed as a SEooC[7], where it had previously been developed in-context. SEooC requirements may need different attributes, and some refactoring work will be necessary within the requirements to introduce a split between in-context development and out-of-context development. The requirements managers together with the engineers can perform the refactoring on a separate branch, and merge it after a review. This even works when the project is large and has thousands of requirements.
4. Now the code base (including requirements) looks like as if it had always been developed according to the new business model.

5.3 Configuration as Code

TRLC allows to declare individual types with individual attributes. Our domain-specific language does not impose any standards as to how a requirement type shall look like. For example, an information object might have different attributes than a requirement object. TRLC is a language to define these kind of types. There is no limit to the number of such types that can be defined. Through type extension there is also support for specialised requirement types, for instance security-related requirements might have different attributes than requirements coming from legal laws.

TRLC gives the requirements managers the freedom they need. As seen in the examples from section 4, the type definitions are stored in text files. This way they are easy to refactor. This is what we call “Configuration as Code”.

These files are under version control in the repository, same as the files containing the content. Being able to merge a configuration update together with a content update in an atomic step is an important enabler to refactor quickly while having a shippable product at every point in time.

[7] safety-element out of context, see ISO 26262

5.4 Examples for Configuration as Code

5.4.1 Mass Edit

Imagine a new external party will be responsible for testing a sub-set of the requirements to solve a resource bottleneck problem. Previously the testing was done in-house by the feature teams themselves. Most likely the requirements manager did not think of such a scenario upfront when they designed the requirements engineering process, and the established attributes don't allow to manage this new business model well. Suddenly they need to introduce a test responsibility into the requirements. A refactoring of the attributes is required, and initial test responsibilities must be added to each requirement. Most likely a script is needed to set the initial values. This is what we call a "mass edit" situation. Many hundreds of requirements might need to be touched. This is very easy to achieve with our Python interface for TRLC, that allows custom scripts to be developed, that can operate on the model. The solution is to branch off, create the necessary changes, review the changes together with architects and business managers, and merge them back into the main branch.

5.4.2 Feature Team Request

Usually feature teams know best what they need. And they might know it better than their requirements managers. Putting the requirements configuration under version control enables feature teams to demonstrate their needs to the requirements managers easily by creating a pull request which contains the desired changes. The requirements managers can then decide whether the proposed solution will be a benefit for all teams working in the project, or whether the team at hand is exposed to a special situation and needs a special requirements configuration and process.

5.5 Continuous Integration

We highly recommend to use a CI system which ensures completeness[8] of the requirements as far as possible. Confirming completeness requires human judgement in the end, but many aspects can be checked with validation rules and by implementing custom checks via scripts. In this section we present a few examples of such completeness checks that a CI can perform for every new pull request:

ASIL Decomposition The functional safety manager of a product might define rules about when and how ASIL[9] decomposition is allowed. A check could look at the trace

[8] Completeness here means no further work is necessary. The requirements work is done.

[9] Automotive Safety Integrity Level as in ISO 26262, but the concept is obviously also applicable to SIL from IEC 61508

between two requirements and forbid a merge to the main branch if the derived requirement has a deviating ASIL compared to the higher-level requirement.

Mandatory Attributes Most attributes for a requirement are mandatory, but some are optional under certain situations. A check could evaluate the conditions and forbid a merge to the main branch if attributes are missing. For example, a rationale must be given in case of safety-related requirements.

Export Consistency Requirements may reference other requirements in their text body or through attributes. If such a requirement is exported to a supplier, then a check could verify if the referenced requirement is also marked to be exported. Otherwise a merge will be prevented by the CI system.

Release consistency Often a set of requirements is used for multiple releases of a software, where single requirements are applicable to selected releases only. In analogy to the above check regarding export consistency, the consistency of the requirements must be checked for each release individually. The CI system can easily generate a report for each release and allow to merge the requirements only if they are consistent for every release.

Traceability Requirements are not the only artefacts in a software development project. Apart from traceability between requirements, the CI can check that the linkage to or from the following artefacts exists:

- software design and architecture
- bus signal definitions
- security analysis
- safety analysis, FMEA[10], HARA[11], HAZOP[12]
- release management
- test cases or proofs
- source code
- argumentation

[10] Failure Mode and Effects Analysis
[11] Hazard Analysis and Risk Assessment
[12] Hazard and Operability Study

To illustrate the power of such automated checks, think about a new feature request that requires to change the ASIL from QM[13] to some higher value. The checks, especially the ASIL decomposition check, now basically behave as an impact analysis and help to estimate the development costs of the new feature. The time a requirements engineer needs to update all the affected requirements until all checks signal a green light is a very good indicator as to how long it will take to implement the change in source code.

Note that applying rules ensures that many inconsistencies are detected even before the requirements go into the main branch. Ideally one does not need a status attribute for requirements, because the main branch contains only valid requirements by definition.

6 Requirements Engineering Process Development

Every project has to embed the TRLC approach as well as the corresponding tooling into their requirements engineering process. Methodology decisions must be made, like how to identify requirements that must be reviewed by a security expert, or how to link requirements with the software design.

TRLC was designed to speed up the initial phase of a project, where topics like above are discussed and decided. It is our clear recommendation to take decisions fast, start developing the product, and refactor the TRLC configuration later, as soon as more information is available.

TRLC was also designed to measure key performance indicators (KPIs) from the beginning. A requirements manager can use our Python API to access all TRLC objects immediately, and measure the KPIs right after the first merge to the main branch. At the beginning the script to measure the KPIs might need frequent changes, just like the type definitions. The alternative is to create a prototype of the tool chain first, and start the product development afterwards once the tool chain is available. This traditional approach is not suited for a volatile, uncertain, complex and ambiguous environment, though, as it is simply too slow until a shippable product is available.

It is important to emphasise that the TRLC framework does not provide any guidance on how the requirements engineering process actually looks like. It also does not impose any restrictions. Process development is still required as part of every project setup. The benefit of using the TRLC framework is the ability to implement every process in an agile way and being able to react on changes appropriately.

7 Limitations and Future Work

Like any approach, there are limitations. One in particular that should be highlighted is that of non-technical users. TRLC is more powerful and automates many things that were

[13] Quality Managed, which is the lowest level in ISO 26262

painful before, but users not happy with using e.g. git will likely have similar issues. There are two mitigating factors however:

- First, reading requirements should not be any harder than it was before, as it is possible to e.g. generate a more readable HTML or PDF document. This is something we're prototyping with TRLC itself: the LRM of TRLC will be a set of TRLC requirements, and a script will convert this to a nicer HTML document.
- Second, it is possible to separate the worlds (in the spirit of using the best tool for each job): high level requirements could be kept in a traditional tool where non-technical feature designers write requirements; and software and component requirements can be expressed in TRCL. In fact this is the most likely scenario for our use-case at BMW, and the two worlds can be linked together using LOBSTER.

We expect to develop the language in a backwards compatible way as various new use-cases arise, and similarly maintain the tooling. On the road map are various IDE or editor integrations for the language (e.g. EMACS or VS Code); a linter for checking consistency of the checks themselves; and some performance improvements.

Once we have used the tools ourselves over a longer period of time on more projects, we intend to publish metrics and findings.

8 Summary and Conclusion

In this paper, we presented the TRLC framework, which was developed to bridge the gap between agile software development and requirements engineering for safety-critical software products. We explained the background and our motivation for such an approach, as well as the key elements of the TRLC framework, including its integration in continuous integration, as well as the corresponding software process development. We highlighted the importance of TRLC to a better requirements traceability process.

In a nutshell, TRLC is a framework including a lean yet powerful domain-specific language that comes along with a free software tool chain and Python API. It shrinks the project start-up phase (with respect to the requirements management process) to a couple of days, so that the product development can start quickly, without neglecting the crucial aspect of requirements engineering. This also holds in a safety-related environment where ISO 26262 must be adhered to. Software projects are not bound to heavy-weight tool chains anymore, whose "external database" approach collides with fast iterations required in agile software development. Instead they can easily build up their own requirements engineering configuration, as well as light-weight supporting tools for validation, KPI calculation, as well as all kinds of data transformation. Depending on the company's size and the amount of projects it is managing at the same time, design patterns as well as guidelines and other templates for TRLC can be shared within the requirements engineering community, to speed up the project setup even more.

Bibliography

[Al09] Allmann, C.: Situations- und szenariobasiertes Anforderungsmanagement in der automotive Elektronikentwicklung. Audi Dissertationsreihe. Cuvillier Verlag, 2009.

[Be01] Beck, Kent; Beedle, Mike; Van Bennekum, Arie; Cockburn, Alistair; Cunningham, Ward; Fowler, Martin; Grenning, James; Highsmith, Jim; Hunt, Andrew; Jeffries, Ron et al.: , Manifesto for agile software development, 2001.

[Br06] Broy, Manfred: Challenges in automotive software engineering. In: Proceedings of the 28th international conference on Software engineering. ACM, pp. 33–42, 2006.

[Fü10] Fürst, Simon: Challenges in the design of automotive software. In: Proceedings of the Conference on Design, Automation and Test in Europe. European Design and Automation Association, pp. 256–258, 2010.

[Ga18] Galinier, Florian: A DSL for requirements in the context of a seamless approach. In: 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 932–935, 09 2018.

[IS02] ISO/IEC JTC 1/SC 22: Z formal specification notation - Syntax, type system and semantics. Standard, International Organization for Standardization, 2002.

[IS18] ISO/TC 22/SC 32: Road vehicles - Functional safety - Part 6: Product development at the software level. Technical report, International Organization for Standardization, 2018.

[Kn86] Knuth, Donald Ervin: METAFONT: the program. Addison-Wesley Longman Publishing Co., Inc., 1986.

[LB15] Leuser, Jörg; Ballhause, Christoph: Anforderungen programmieren - eine domänenspezifische Sprache (DSL) im Praxiseinsatz. OBJEKTspektrum, Online special Requirements Engineering, 6 2015.

[Mi21] Mihailovici, Marius: , Wenn Software Software Schreibt. Porsche Engineering Magazin, 1 2021.

[SBS20] Schlichthaerle, Stefan; Becker, Klaus; Sperber, Sebastian: A Domain-Specific Language Based Architecture Modeling Approach for Safety Critical Automotive Software Systems. In: Software Engineering (Workshops). 2020.

[Sh12] Shen, Mengjiao; Yang, Wenrong; Rong, Guoping; Shao, Dong: Applying agile methods to embedded software development: A systematic review. In: 2012 Second International Workshop on Software Engineering for Embedded Systems (SEES). pp. 30–36, 2012.

[VD17] VDA QMC Working Group 13 / Automotive SIG: Automotive SPICE Process Assessment / Reference Model. Technical report, VDA QMC), 2017.