Towards using Language Workbenches for Critical Software

Markus Voelter
independent/itemis
voelter@acm.org

Daniel Ratiu
Siemens AG
daniel.ratiu@siemens.com

Bernd Kolb, Klaus Birken
itemis AG
{kolb|birken}@itemis.de

Patrick Alff
Voluntis
Patrick.Alff@voluntis.com

Andreas Wortmann
OHB System AG
andreas.wortmann@ohb.de

Arne Nordmann
Bosch Corporate Research
arne.nordmann@de.bosch.com

Abstract

Language workbenches allow developers to create, integrate and efficiently use domain-specific languages (DSLs), typically by generating programming language code from models expressed with DSLs. This leads to increased productivity and higher quality. However, in safety-/mission-critical environments, such generated code may not be considered trustworthy, preventing the use of language workbenches. In this paper, we propose an approach to still use such tools in critical environments. We argue that DSL-based models are easier to validate, and that the additional risk resulting from the transformation to code can be mitigated by an suitably designed transformation and verification architecture. We round out the discussion by evaluating the degree to which this approach is appropriate for critical software in space, medical, automotive and robotics systems.

1. Introduction

In critical systems (we refer to both civilian safety-critical and military mission-critical systems as *critical systems*), hardware and software components require a higher level of trust compared to a non-critical context because system failures may lead to financial loss (banking), loss of non-replaceable systems (space), environmental damage (power plants) or loss of life (medical).

The higher a system's criticality, the more confidence must be provided regarding its proper functioning. Confidence can be built by architectural means (such as redundancies) in the system itself and by following a defined development process. The latter includes *tools*, software programs used in constructing the system. It has to be ensured and documented that the use of those tools does not lead to additional errors in a critical software component (CSC).

Development of critical systems is governed by standards; all of them are justifiably conservative. For example, they require the use of well defined, unambiguous language subsets of C or Ada or proven model-driven development tools like Matlab Simulink. Defining custom DSLs is, at first glance, at odds with this conservative perspective. However, there are also benefits of using DSLs and code generation, in particular for validation. This paper describes this conflict and proposes approaches to reduce it.

Contribution We identify the challenge in using DSLs and language workbenches (LWBs) for critical systems (Sections 2 and 3) and then present an architecture for the use of LWBs for developing critical software (Sections 4, 5 and 6). It assumes that one cannot trust the correctness of a DSL and the LWB, which is why we introduce mitigations for the introduced risks. The paper also contains outlines of how this architecture can be legally used in different domains (Section 8) and how the resulting efforts can be justified (Section 7). The paper describes work in progress. It is intended to bring the safety and language communities together to reduce the tension between safety and productivity.

2. Background

2.1 Safety and Standards

Domains and Standards Critical systems are found in domains such as railway, medical, robotics, aerospace or automotive. Each has different regulatory bodies, different organizational and cost structures and different development philosophies; for example, because of the importance of unit cost, cars are developed differently from airplanes. These differences are captured in each domain's safety standards such as DO-178C for aviation or ISO26262 for automotive systems. However, these standards all embody the philosophies expressed in the generic safety standard IEC61508.

, .
Copyright © ACM [to be supplied]. . . $15.00.
http://dx.doi.org/10.1145/

Tools The standards describe requirements for the tools used in the development of critical systems. Since this paper focuses only on critical *software* development, three categories of tools are relevant: Development tools (e.g., compilers) create artifacts that execute as part of a CSC. Analysis tools (e.g., static analyzers) ensure some aspect of correctness of the CSC. Management tools manage data (e.g., requirements). Development tools, of which LWBs and DSLs are examples, imply the biggest risk, because they may introduce *additional systematic errors* into the CSC if the generators are faulty. The standards of all domains require the mitigation of such errors.

Tool Reliability and Mitigation Strategies Some development tools, such as SCADE[1], can be *assumed* to work correctly and to not introduce errors into a CSC; no mitigations must be put in place. Such tools are called *qualified* tools. Each domain standard has specific ways of qualifying a tool, but three general approaches can be distinguished: (1) Provide proof and/or extensive validation that the tool is correct. (2) The tool itself is developed with a process that follows a safety standard. (3) A specific version of a tool has been "proven in use", which means that it has been used in similar projects for a long time, reports about malfunctioning of the tool have been collected and specific mitigations are put in place.

However, none of these approaches can be used for a DSL developed in a LWB because (1) existing LWBs have not been developed with qualified tools or a safety-process; (2) they have not been used for years for the development of critical systems and are thus not proven in use, and (3) proving the LWB or the DSLs correct is usually not feasible. Thus, mitigation strategies must be used in *how the tool is used*. This paper introduces a set of such mitigations.

2.2 Terminology

Verification vs. Validation *Verification* ensures that the software works without intrinsic faults. It verifies inherent properties through test cases or analyses that verify previously defined properties or check for general weaknesses in the code, such as uninitialized reads or invalid dereferencing of pointers. Verification is performed by software developers. *Validation* ensures that the software does what the requirements specify. Example validation activities include requirements review, simulation, acceptance tests or tracing to requirements. Not all of these can be automated and some of them are performed by stakeholders that are not developers.

Testing vs. Static Analysis A *test case* exercises the CSC through its (test-specific) APIs, asserting whether it reacts correctly. *Static analysis* does not run the program, instead it analyses program code for a *class* of faults, possibly relative to previously specified properties. Examples include checking the satisfiability of certain conditions, checking temporal properties on state machines or using abstract interpretation for ruling out runtime errors such as division by zero.

[1] `http://www.esterel-technologies.com/products/scade-suite/`

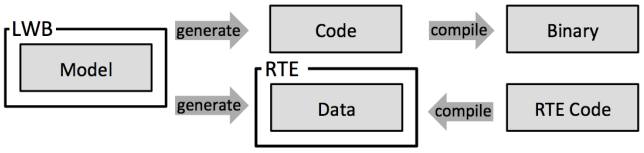

Figure 1. From the model we generate source code, which is then compiled to a binary and executed. Alternatively, we generate data that is interpreted and executed by a runtime environment (RTE), which is built from its own source code.

Coverage Both testing and static analysis suffer from the coverage problem: a fault may not be detected if the engineer fails to write a test case or specify a verification property. As a remedy, code reviews may find that some tests/properties are missing, various kinds of coverage [32] may be measured by tools, or test case generation may automatically generate test cases that raise coverage to a required level.

3. Motivation and Problem

A LWB can be used to define languages that help describe an CSC; code generation is used to create the final, executable artifact (see Figure 3). The executable artifact is either source code that is then compiled, or data (e.g., XML) that is interpreted by a runtime environment (RTE).

Benefits Because the language used for modeling the CSC in the LWB is custom-built for the particular domain (a domain-specific language, or DSL), the description of the CSC using the DSL is more concise, more readable and more analyzable than the corresponding low-level code.

> Verification and validation is more efficient than on code level. Potentially, non-programmer stakeholders can read, or even write the DSL code, integrating them more directly into the development process (see Section 4).

If the transformation to the executable code is correct, this leads to significant gains in productivity through correctness-by-construction [4].

Problem However, the generation step might be faulty, introducing errors into the CSC. This can be mitigated by performing verification and validation on the generated artifacts, essentially ignoring the models for the purpose of building confidence in the system. Due to the high effort involved, this voids many of the aforementioned benefits and is thus of limited value. The challenge thus becomes:

> How can a non-qualified LWB and custom-built DSLs be used in the development of critical systems without (manually) performing all verification and validation activities on the generated artifacts, thereby loosing the benefits of using the LWB and DSLs in the first place?

4. The Benefits of DSLs and LWBs

Our argumentation relies on the claim that it is desirable to perform as many development activities as possible on a suitably abstract model. While we provide some rationales, we assume that the reader accepts most of these points, based on their own experience and the extensively documented productivity benefits of DSLs. We summarize below.

Implementation Effort A DSL can reduce the implementation effort because a it is typically more concise and requires less boiler-plate code because of its more appropriate abstractions. Low-level mistakes (e.g., faulty pointer arithmetics in C) cannot be made. IDE support can also be better because the semantics of the DSL is known by the IDE. Note that the implementation is not where most of the effort is spent in critical software; it is spent in validation and verification. However, taken together with the validation and verification advantages outlines below, a more efficient implementation allows more and faster iterations, thus significantly contributing to overall efficiency.

Verification and Test Models expressed with a suitable DSL avoid the need to "reverse-engineer" domain semantics from low-level implementation code, simplifying verification and test. For example, if state machines are represented first-class (as opposed to, e.g., `switch`-statements), an automated analysis to detect dead or unreachable states is much simpler to perform (and hence, to implement).

Verification properties or test cases can also be expressed at the higher level of abstraction: properties about the state machine can directly refer to states and events, and test cases can explicitly trigger events and assert states. Verification results can be reported at the level of the domain abstractions.

Clearer semantics are also useful for test case generation. For example, variables can be annotated with ranges or other more complex constraints; the test case generator can use those as the boundaries for the tests (instead of the generic `MAX_INT` and `MIN_INT` boundaries).

Validation Models can be used to front load [20] validation, reducing the cost of errors [2]. They can be simulated and tested, to ensure that they behave correctly – so-called model-in-the-loop testing. Validation also involves reviews, either by other developers or by QA people. Models that use appropriate abstractions and notations make the review more efficient because they are easier to comprehend and easier to relate to requirements (the semantic gap is narrower). [17] confirms empirically that program comprehension is improved with DSLs. For validation on model level to work, the semantics of the DSL must be clear to everybody involved: we briefly discuss this issue at the beginning of Section 6. In general, non-programmer stakeholders (such as systems engineers, medical professionals or space scientists) can be integrated earlier and more efficiently.

Finally, tracing of design, implementation and test artifacts to requirements can be more easily supported on models than on code (a practical issue with current tools).

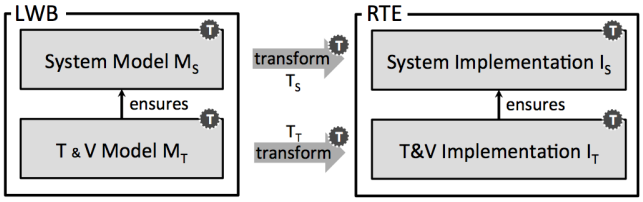

Figure 2. Baseline tool architecture: fundamentally, all tests or verifications (T&V) are expressed at the model level (because this is more convenient and/or productive). However, they are executed at the level of the implementation.

Derivation of Artifacts In critical domains, loads of documents are required as evidence for the correct functioning of to demonstrate the adherence to the development process. Generating these documents from models ensures consistency with the actual system and further reduces effort. Examples include diagrams representing the structure or behavior of the system as well as trace reports.

5. Connecting to the Code

Execution of the CSC happens on the implementation (generated code or interpreted in an RTE, see Figure 3). Due to the abstraction gap between models and the low-level code, big-step vs. small-step semantics [9] phenomena might occur when the generated code exhibits behaviours not visible at the model level. Our challenge thus becomes:

> How can we assure that the generated code behaves in exactly the way the validated model prescribes?

Sections 5.1 and 5.2 propose an architecture that answers this question. We also discuss steps to ensure the correctness of the executing code in Section 5.3 that would also be taken for manually written code; we include them primarily for completeness. Section 6 outlines additional safety mechanisms that are not directly related to code generation or the correctness of the generated code.

5.1 General Architecture

As shown in Figure 2, we model the CSC as well as test cases and verification properties with DSLs in an LWB. We then transform both to their executable representation. Tests can be executed at the model level (e.g., through an interpreter) and at the code level (by generating them to the implementation level; software-in-the-loop testing). Similarly, some verification can be performed both at the model and code level. An example is model checking, which can be done by translation the model to a tool such as Z3 (note that in this case, this transformation also has to be assured) but also at the code level (by encoding the properties in C code as shown in [22] and [25]). Traces connect models to requirements; the transformations propagate the traces to the implementation to connect it to requirements, too. Trace reports are generated to demonstrate requirements coverage.

We now analyze the risks of this architecture in terms of a faulty CSC implementation, and introduce mitigations. Figure 3 shows the final architecture with mitigations included.

5.2 Assuring the Transformation

We introduce means to ensure that models M_S and M_T are transformed correctly to the implementation code I_S and I_T by transformations T_S and T_T. We assume that the source models are correctly validated and verified; see Section 4.

T_S has an error and generates some behaviors wrongly.
Ex: A `switch` statement is generated from a state machine, but `break` statements are missing.

Since the tests in M_T test the model correctly, and since T_T is correct, the tests in I_T, will fail and detect the error in T_S.

T_T has an error and generates some tests wrongly.
Ex: The computation of certain expressions in `assert` statements ignores an exception and defaults to true.

Since the tests in M_T test the model correctly, and since T_S is correct, the tests in I_S will fail and detect the error in T_T.

T_T has a systematic error that results in all (or some class of) tests generating wrongly.
Ex: The developer has forgotten to change the `assert true;` example code to an actual template. Every tests will succeed.

This can be detected by manually reviewing some test cases in the generated code or by fuzzing some of the test cases in M_T; they would then have to fail.

> *Some test cases should be reviewed in their generated version. Fuzzing should be used to prevent systematic errors in test cases.*

Both T_S and T_T have unrelated errors.
Ex: A mix of the above examples.

If the errors are truly unrelated, then the first two cases apply at the same time and thus some tests will fail; inspection will reveal the unrelated errors in T_S and T_T so they can be fixed.

T_S and T_T have unrelated, but compensating errors.
Ex: T_S translates actions in hierarchical state machines wrongly (entry-transition-exit instead of exit-transition-entry). The generator for M_T translates lists of assertions in reverse order.

Unrelated compensating errors are exceedingly unlikely (cf. the contrived example) but cannot be fully mitigated.

> *The residual risk can be reduced by increasing the number of test cases and ensuring a degree of overlap.*

T_S and T_T have related errors.
Ex: Both transformations generate `int8`s instead of `int16`s.

The main cause for related errors is that T_T and T_S have been implemented by the same developer, who has made a common, conceptual mistake in both transformations (such as misinterpreting a requirement).

> *The transformations T_S and T_T should be implemented by different developers.*

Wrong requirements can also lead to wrongly implemented functionality in M_S and wrongly, but consistently, tested in M_T. However, this is not a problem of the transformations T_S or T_T, but of the two models; they can be found out by validating the model M_T and M_S (see Section 4). Consider having M_S and M_T being created by different developers.

The transformation engine itself has an error.
Ex: Polymorphic dispatch in transformation rules is faulty, applying the wrong transformation rule.

While this is unlikely for tools that have been used for years, mitigation of this risk might be required nonetheless:

> *If you don't trust the transformation engine, develop redundant T_S and T_T with a different transformation technology. The tests must succeed there as well.*

The tests from M_T can then be executed on both implementations. This possibility is a huge advantage of describing the tests in a model in the first place.

5.3 Low-Level Code Assurance

We discuss additional steps that are similar to what would be done in manually written code. However, we point out specific advantages resulting from the generative approach.

Failures because of the implementation code structure
Ex: Stack overflows, numeric precision errors, timing violations, or invalid pointer derefencings.

To prevent low-level failures as a consequence of the small-step semantics of the execution platform, make sure that all code paths of the implementation are executed. This can be achieved by measuring coverage on M_S. If failures occur, the transformation T_S must be fixed – likely fixing all errors that were a consequence of this faulty transformation.

> *Measure coverage on I_S, on the target platform, and ensure near-100% coverage for the tests.*

For some languages, static analysis tools that proof the absence of some classes of errors are available (such as Astree or Polyspace for C). The generated code can use patterns that simplify the analysis, and semantic annotations can be added the enable more meaningful analysis (an example is Frama-C Jessie [7]). The semantic information is available in M_S and can be mapped by T_S. An example of this approach to verify concurrent C programs is given in [8].

> *Use static analysis tools to further increase the quality of the generated code; generate analyzable patterns via T_S and add semantic annotations based on M_S.*

The system will typically be run on a target device that is different from the developer's computer. The implementation-level tests must be executed on this target device, using that device's compiler, because the hardware/compiler might introduce additional errors. This is referred to as hardware/software-in-the-loop testing.

Insufficient resources may lead to errors.

Ex: A data queue associated with a sensor overflows because of an unexpectedly high signal rate on the sensor; data is lost.

The target environment may be restricted in terms of memory, processing power or other critical resources, and the program may fail because it runs out of resources.

> *Run the tests cases I_T on the real target hardware, with real (amounts of) data.*

Capturing the expected resource utilisation in M_S and then validating at runtime helps with diagnostics – resource starvation is hard to debug otherwise.

The implementation may be exploited maliciously.

Ex: Adversaries intentionally supply too much data, making a buffer run over its limits.

Because of the degrees of freedom involved in implementation code, the system may be attacked by exploiting those degrees of freedom. Those cannot be predicted from the model level. Hence, penetration testing [1] on implementation level is unavoidable.

> *Perform penetration tests on I_S to ensure the absence of attack vectors.*

The generator can generate potentially more secure code [30], e.g. by calling sanitizing functions for all inputs [23].

The target compiler or runtime system may be buggy.

Ex: The interpreter deals wrongly with operator precedence.

In the very rare case when the target infrastructure has not yet been proven in use, this risk must also be mitigated. One way of doing this is is to compile/run the same I_S/I_T on different target infrastructures (e.g., different C compilers).

> *Execute the program in different target environments to find problems in the target environment.*

If no two different target environments for the same I_S/I_T are available, then you have to create different I_{S2}/I_{T2} via additional T_S and T_T. Test divergence may then also be a consequence of errors in one of the T_S/T_T.

A special case of this approach is to treat the simulation environment (for end user experimentation) as one of the two execution targets. This way, no additional effort (beyond the simulator itself) is required. However, in this case (and in all other cases where the two execution platforms are significantly different), non-functional requirements such as timing or resource contention cannot be assured.

6. Additional Safety Mechanisms

In addition to the risks identified in this paper, many other things can go wrong in critical systems; many of those mentioned in [16] are relevant. In this section we discuss a few additional concerns that are not directly related to the transformations, but still relate to LWBs and DSLs.

Defined Language Users of the DSLs have to understand its semantics to be able to create correct M_S and M_T models. The same is true for the developers of T_S and T_T so they can

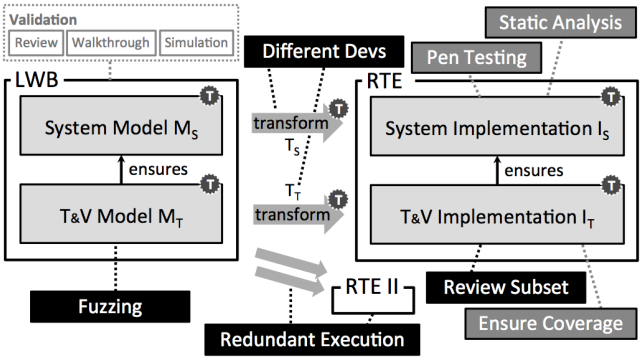

Figure 3. Annotated baseline architecture from Figure 2. Black boxes represent code verification activities that are necessary because LWBs and code generation are used; grey boxes are verification activities that would be done in the same way if the code was written manually; and white boxes are activities to validate the quality of the model. The circled Ts represent traces to requirements.

"implement" these semantics in I_S and I_T. To this end, a clear definition of the language and its semantics is needed.

QA'ing the Language This paper presupposes that we cannot reliably assure the correctness of the DSLs and the transformations (even though there is some work on these aspects [19, 28]), which is why we propose an architecture that remedies the consequent risks. Nonetheless, the DSLs *should* be systematically verified. [24] discusses language testing in MPS, based on manually written type system tests, automatically generated test cases for language structure and syntax, and measuring transformation coverage.

Quality of the Generated Code Some domains require code to conform to certain guidelines. For example, C code in automotive systems is expected to conform to MISRA-C [21] to improve readability and to prevent the use of language constructs that are hard to understand (and analyze), lead to unspecified behaviour or trigger known errors in the compiler (see Section 5.3, *Target compiler may be buggy*). Compliance can be checked by tools, e.g. the one from LDRA[2]. Generating code that conforms to such standards is easy if the transformation developer knows the guidelines. The importance of the guidelines is reduced for generated code: e.g., MISRA-C reduces the strictness of some rules for generated code. Anecdotally, we have been granted MISRA exceptions for generated polymorphic dispatch code that uses (otherwise prohibited) void pointers.

Architectural Mechanisms In this paper, we discuss how the implementation can be guaranteed, to some degree, to be correct. An orthogonal approach is runtime monitoring and fault detection [12], which transitions the system into a safe state if a fault is detected. In medical systems, such a

[2] http://www.ldra.com/en/software-quality-test-tools/group/by-coding-standard/misra-c-c

state may be to tell the user to call a doctor and disregard the software. Examples of such architectural mechanisms are checksums (to detect random bitflips), redundant sensors (to detect faulty sensors), monitoring of timing or resource consumption (to detect emerging resource contention), or separately specified validation rules for data or behavior (similar to tests, specified separately by a separate developer, to avoid common cause errors). The implementation of some of these mechanisms can be automated through the transformations, leading to a reusable safety-aware platform for specific domains (an example for avionics is presented in [10]).

Safety Analysis Models LWBs can also be used to explicitly model safety analyses, as shown in [26] or [3]. By connecting fault tree analyses (FTAs) or failure mode and effects analyses (FMEAs) directly with system and test models (M_S/M_T) expressed in the same LWB, benefits can be achieved compared to using separate, external tools.

7. Justification of Higher Efforts

The additional verification and validation steps introduced in Section 5.2 and summarized in Figure 3 lead to additional effort compared to the use of LWBs outside of critical systems. However, the critical systems community accepts that their verification effort is several factors higher than in other systems (up to 1,000 USD per line of code total cost [13] compared to 15-50 USD in regular embedded systems [15]), so, in general, higher effort is acceptable. However, it has to be weighed against the traditional, manual process, taking into account the benefits discussed in Section 4. Anecdotally, the introduction of a LWB and DSLs can reduce the development effort by a factor of 10 or more. Since most of the QA mechanisms introduced in Section 5 can be automated (and hence, are one-time costs), the fundamental benefits of LWBs and DSLs are not compromised; however, the threshold at which the approach becomes viable may be higher than in non-critical software.

8. Mapping to Domains

The standards relevant for the various application domains typically do not consider LWBs and DSLs or they may require more or different activities. This section outlines the standards relevant to four domains, plus the additional activities required in each domain.

Health and Medicine/FDA Using software in a medical context in the US requires authorization from the FDA. The FDA defines notion of a software medical device, which is a software-only solution for diagnosing or treating diseases. The relevant FDA documents[3] classify a software component according to a level of concern; the algorithmic core of software medical devices, is classified as *high* and the FDA requires a hazard analysis, basic documentation, hazard mitigations, a description and justification of residual risk as well as so-called special documentation.

Basic documentation encompasses a description of what the software is, the hardware it requires, how the end user is guided, as well as a discussion of QA and maintenance processes. The *hazard analysis* encompasses a list of all potential hazards, their estimated severity and possible causes. The *hazard mitigations* then describe how the design of the system migitates these hazards, including protective measures, user warnings, user manuals or special labelling materials. As discussed in Section 5, LWBs and DSLs introduce additional hazards. We also show in Section 5 how these are mitigated. We are confident that these mitigations are sufficient, so performing these mitigations and documenting the process and the results should be a workable approach. The required *justification of residual* risk should not contain any significant risks that stem from this approach.

Software with a high level of concern also requires *special documentation*, i.e., assurances that the development process is appropriate and sufficient for the intended use. This includes systematic management of the requirements for the DSLs, tracing of requirements to the DSL implementation, as well as the well-definedness of the language and testing of its implementation (see *Defined Language* and *Language QA* in Section 6). This is significant additional effort, but it is effort that has to be performed only once (and if the language evolves). Therefore, it can be justified considering the fact that the number of systems developed with the language will grow over time, as argued in Section 7.

Considering that the FDA has found that the majority of software-related device failures are due to design errors[4] and the most common problem was failure to validate software prior to routine production, we are confident that the benefits of inducting DSLs and LWBs will outweigh their implementation risks and thus produce an excellent risk-benefit ratio, accelerating the production scale-up and reducing the cost of QA of medical device software.

Automotive Major trends in automotive software (e.g., integration on fewer but more powerful computers, advanced driver assistance, and autonomous driving) lead to a larger amount of software at higher safety levels. The relevant standard, ISO26262 *Road vehicles – Functional Safety* classifies software according to risk (ASILs A, B, C and D) and assigns appropriate safety measures.

Model-based development tools are well-established (e.g., AUTOSAR[5] for generating component interaction code and Matlab/Simulink for generating component implementations). ISO26262 explicitly acknowledges model-based development for simulation and code generation (Annex B: *The seamless utilisation of models facilitates a highly consistent and efficient development*). The safety of model-

[3] *Guidance for Industry, FDA Reviewers and Compliance on Off-The-Shelf Software Use in Medical Devices*; September 1999

[4] UCM263366: Mobile Medical Applications — Guidance for FDA staff

[5] `http://autosar.org`

based development tools is addressed by proven-in-use arguments and by regarding generated code as manually written.

ISO26262 also specifies that, for untrusted tools, there must be "very high confidence" that errors are detected. Tool vendors provide reference workflows (e.g., [6]) that define activities around the modeling/generator-toolchain to satisfy this requirement. Most activities defined by these workflows are covered by the LWB approach as described in Section 5, e.g., module and integration tests on model level. Checking of modeling guidelines is necessary for a general-purpose modeling tool, but not needed with an LWB approach as DSLs will enforce guidelines directly. Automatic test case generation on model level can be applied in LWBs to avoid tedious manual creation of test cases. As DSLs cannot be proven-in-use by definition, some additional measures are needed compared to the state-of-the-art reference approach described in [6]. Examples include the aforementioned ensuring of diversity by developing transformations T_S and T_T separately, and by deploying on two different RTEs.

Summing up, we assume that there is a realistic chance that the approach proposed in this paper will work. We will verify this in upcoming projects.

Space/ESA The ESA-founded European Cooperation for Space Standardization[6] (ECSS) is an initiative to develop a coherent, single set of standards for all European space activities. The ECSS-Q-ST-80C standard addresses software product assurance, including qualification steps to be agreed between the customer (often ESA) and the supplier. Since most European space projects focus on unmanned satellites and probes, the software is considered mission-critical rather than safety-critical, allowing for considerable relaxation.

ECSS-Q-ST-80C has a notion of generated code, traditionally modeled using tools like Simulink (for control loops), or UML tools (for class skeletons). The code is qualified either by using a qualified code generator (Simulink and its code generators are considered proven in use) or by treating the generated code as manually written code.

As far as we know, DSLs and LWBs have not been used in flight software development. Since the qualification criteria are tailored specifically for each mission, the approach described in Section 5 should be sufficiently convincing for a customer to accept. Nevertheless, a comprehensive demonstration based on a representative prototype is necessary.

Robotics From the regulatory point of view, traditional industrial manufacturing robots are still largely treated as machines and their safety is evaluated through a safety risk assessment based on ISO12100:2010 or IEC61508.

For development tools and programming languages, IEC61508 highly recommends the use of certified tools and translators, as well as relying on trusted and verified software components. However, IEC61508 does not clearly define the criteria for a translator to be accepted as *certified*, hence the means discussed in Section 5.2 to assure correct transformations appear to be suitable.

IEC61508 also states that the chosen language should be restricted to unambiguously defined features, match the characteristics of the application, and contain features that facilitate the detection of errors [11]; properties that fit well with the use of DSLs and LWBs (Section 4).

Avoidance of collisions between robots and humans is at the core of robot safety, regulated by DIN/EN/ISO10218. Therefore, classical industrial robotics come with safety fences, which tightly constrain the possibility of harming humans. However, modern robots might be required to intentionally make contact with humans (teaching, collaboration) in open, unstructured environments where testing cannot be exhaustive and cannot be cost effective: safety must be an assured also for unexpected situations. A promising approach is the use of formal methods to reduce the test effort. For those to be adopted more widely across industry, they need to be flexible and easy to use. DSLs and LWBs are a good foundation for formal methods, analysis, and even proofs, based on domain-specific abstractions. Section 5.1 discusses an architecture that seems suitable to enable easier access to domain expert, and therefore reducing time and cost of adoption, as discussed in Section 3.

9. Related Work

Overviews A general comparison of safety standards for different domains is presented in [18]. While it does not directly consider DSLs, LWBs or code generation, it provides a good overview over the general challenges. [5] focuses on the challenges of using models and code generation (with an MDA flavor) for safety-critical systems. Among other things, they identify that “program compilers or interpreters must be assured somehow” – which is the challenge we address in the current paper.

Examples DSLs have been used in other critical domains. For example, [27] discusses the use domain of operating system driver development. Haxthausen and Peleske [14] describe a DSL for defining the interlocking protocol of railway systems. Both papers exploit the benefits of better analyzability on DSL level and identify the need for verifying the interpreter/generator and the compiler. Both do not address this challenge. however. This clearly emphasizes the need for what we discuss in this paper.

Formalisation [31] defines the formal requirements of DSLs that use formalisms from automata theory. A similar approach of using formal foundations for the DSLs themselves seems to be used by Functor Scalor[7] even though we could not find any technical papers. This approach is orthogonal to ours in the sense that even provably correct models must be translated correctly to executable source code. Several projects address the correctness of language definitions and their implementations by using more analyzable for-

[6] http://www.ecss.nl/

[7] http://www.functor.se/products/scalor/

malisms for specifying the language implementations themselves. The ultimate goal is to prove a language correct relative to certain characteristics. Next to the previously mentioned [19], Visser's essay [29] expressed the goals and possible approaches best.

10. Conclusions and Future Work

We proposed an architecture for using LWBs and DSLs in critical software development. The architecture mitigates the risks of potentially faulty transformations from DSL-based models to the code-level implementation. We outline the degree to which this approach is feasible (or must be modified) for use in robotics, space, automotive and medicine.

Our future work includes running projects in the space, medicine and automotive domains to learn which modifications to the approach are necessary to get it past the certification authorities. We will also collect concrete numbers on the increased efficiency in critical software development using LWBs. One of our customers plans to develop a qualification kit for an embedded software development tool based on MPS; we expect a lot of input for the use of MPS in critical software. Finally, Fortiss is working on integrating provably correct transformations [19] into MPS.

We are confident that this architecture (plus domain-specific additions) will help to make LWBs and DSLs acceptable for use in real-world projects, thus allowing the critical software industry to reap the benefits of these technologies that are documented for non-critical domains.

References

[1] B. Arkin, S. Stender, and G. McGraw. Software penetration testing. *IEEE Security & Privacy*, 3(1):84–87, 2005.

[2] B. W. Boehm et al. *Software engineering economics*, volume 197. Prentice-hall Englewood Cliffs (NJ), 1981.

[3] C. Buckl, M. Regensburger, A. Knoll, and G. Schrott. Models for automatic generation of safety-critical real-time systems. In *ARES 2007 Conference*. IEEE.

[4] R. Chapman. Correctness by construction: a manifesto for high integrity software. In *Proc. of the 10th Australian workshop on Safety critical systems and software*. Australian Computer Soc., 2006.

[5] P. Conmy and R. F. Paige. Challenges when using model driven architecture in the development of safety critical software. In *4th Intl. Workshop on Model-Based Methodologies for Pervasive and Embedded Software, 2007*. IEEE.

[6] M. Conrad. Verification and validation according to iso 26262: A workflow to facilitate the development of high-integrity software. *ERTS2 Conference 2012*.

[7] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c. In *International Conference on Software Engineering and Formal Methods*. Springer, 2012.

[8] M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte. Vcc: Contract-based modular verification of concurrent c. In *ICSE Companion*, 2009.

[9] O. Danvy and K. Millikin. On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion. *Information Processing Letters*, 106(3), 2008.

[10] S. Görke, R. Riebeling, F. Kraus, and R. Reichel. Flexible platform approach for fly-by-wire systems. In *2013 IEEE/AIAA Digital Avionics Systems Conference*. IEEE.

[11] W. A. Halang and J. Zalewski. Programming languages for use in safety-related applications. *Annual Reviews in Control*, 27(1), 2003. .

[12] R. Hanmer. *Patterns for fault tolerant software*. John Wiley, 2013.

[13] B. Hart. Sdr security threats in an open source world. In *Software Defined Radio Conference*, pages 3–5, 2004.

[14] A. E. Haxthausen and J. Peleska. A domain specific language for railway control systems. In *Proc. of the 6thth biennial world conference on integrated design and process technology*, 2002.

[15] P. Koopman. Embedded Software Costs 15–40 per line of code (Update: 25–50). `http://bit.ly/29QHOlo` (URL too long).

[16] P. Koopman. Risk areas in embedded software industry projects. In *2010 Workshop on Embedded Systems Education*. ACM, 2010.

[17] T. Kosar, M. Mernik, and J. C. Carver. Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments. *Empirical software engineering*, 17(3), 2012.

[18] E. Ledinot, J.-M. Astruc, J.-P. Blanquart, P. Baufreton, J.-L. Boulanger, H. Delseny, J. Gassino, G. Ladier, M. Leeman, J. Machrouh, et al. A cross-domain comparison of software development assurance standards. *Proc. of ERTS 2012*.

[19] L. Lúcio, B. Barroca, and V. Amaral. A technique for automatic validation of model transformations. In *MODELS 2010*. Springer.

[20] A. Michailidis, U. Spieth, T. Ringler, B. Hedenetz, and S. Kowalewski. Test front loading in early stages of automotive software development based on autosar. In *DATE 2010*. IEEE.

[21] MISRA. Guidelines for the use of C in critical systems, 2004.

[22] Z. Molotnikov, M. Völter, and D. Ratiu. Automated domain-specific C verification with mbeddr. In *ASE 2014*. ACM.

[23] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *IFIP International Information Security Conference*. Springer, 2005.

[24] D. Ratiu and M. Voelter. Automated Testing of DSL Implementations. In *11th IEEE/ACM International Workshop on Automation of Software Test (AST 2016)*, 2016.

[25] D. Ratiu, B. Schaetz, M. Voelter, and B. Kolb. Language engineering as an enabler for incrementally defined formal analyses. In *1st Int. Workshop on Formal Methods in Software Engineering*. IEEE, 2012.

[26] D. Ratiu, M. Zeller, and L. Killian. Safety. lab: Model-based domain specific tooling for safety argumentation. In *International Conference on Computer Safety, Reliability, and Security*. Springer, 2015.

[27] L. Réveillère, F. Mérillon, C. Consel, R. Marlet, and G. Muller. A dsl approach to improve productivity and safety in device drivers development. In *ASE 2000*. IEEE.

[28] V. Vergu, P. Neron, and E. Visser. Dynsem: A dsl for dynamic semantics specification. Technical report, Delft University of Technology, Software Engineering Research Group, 2015.

[29] E. Visser, G. Wachsmuth, A. Tolmach, P. Neron, V. Vergu, A. Passalaqua, and G. Konat. A language designer's workbench: A one-stop-shop for implementation and verification of language designs. In *Proc. of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. ACM, 2014.

[30] M. Voelter, Z. Molotnikov, and B. Kolb. Towards improving software security using language engineering and mbeddr c.

[31] M. Wasilewski, W. Hasselbring, and D. Nowotka. Defining requirements on domain-specific languages in model-driven software engineering of safety-critical systems. 2013.

[32] M. Weiser, J. D. Gannon, and P. R. McMullin. Comparison of structural test coverage metrics. *IEEE Software*, 2(2):80, 1985.