Using Language Workbenches and Domain-Specific Languages for Safety-Critical Software Development

Markus Voelter
independent / itemis AG
voelter@acm.org

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

Federico Tomassetti
independent
federico@tomassetti.me

Patrick Alff, Laurent Wiart
Voluntis
{Patrick.Alff|Laurent.Wiart}@voluntis.com

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

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

Abstract

Language workbenches support the efficient creation, integration and use of domain-specific languages. Typically, they execute models by code generation to programming language code. This can lead to increased productivity and higher quality. However, in safety-/mission-critical environments, generated code may not be considered trustworthy, because of the lack of trust in the generation mechanisms. This makes it harder to justify the use of language workbenches in such an environment. In this paper we demonstrate an approach to use such tools in critical environments. We argue that models created with domain-specific languages are easier to validate, and that the additional risk resulting from the transformation to code can be mitigated by a suitably designed transformation and verification architecture. We validate the approach with an industrial case study from the healthcare domain. We also discuss the degree to which the approach is appropriate for critical software in space, automotive and robotics systems.

1. Introduction

In safety-critical systems, hardware and software components require a higher level of trust compared to non-critical contexts because system failures may lead to financial loss (finance), loss of non-replaceable systems (space), environmental damage (power plants) or user harm or death (healthcare). The higher a system's criticality, the more confidence must be provided regarding its proper functioning. Confidence can be built by architectural means in the system itself (such as redundancies) and by following particular well-defined development processes. The latter includes *tools*, programs used for constructing the system. It has to be ensured and documented that the use of those tools does not incur additional errors in a critical software component (CSC).

Development of critical systems is governed by standards, specific to the particular domain; all of them are understandably 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 domain-specific languages (DSLs) with specific code generators or interpreters is, at first glance, at odds with this conservative perspective. However, there are also benefits, in particular for validation, which is why it is desirable to use those tools in safety-critical contexts. This paper explains this conflict and demonstrates a practically proven way to overcome it.

Contributions This paper makes four contributions: (1) an analysis of the risks involved in using DSLs and language workbenches (LWBs) regarding the introduction of faults into a CSC, (2) an architecture for mitigating these risks, (3) a case study from the healthcare domain that validates the architecture, and (4) brief discussions of the applicability of the approach to three other safety-critical domains.

Structure We provide some background to our approach and define important terms in Section 2. The advantages of using LWBs and DSLs are recapped in Section 3; this serves as the motivation of why one would want to use DSLs in the first place. We then define the problem associated with the lack of trust in DSLs and generators in Section 4. Our first two contributions, the risk analysis, as well as the mitigations we propose, follows in Section 5. The validation

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

through the detailed case study with Voluntis' medical companion apps is covered in Section 6, the applicability to other domains is briefly discussed in Section 7. We conclude the paper with related work, a discussion and conclusions plus future work in Sections 8 through 10.

2. Background

2.1 Safety, Standards and Tools

Domains and Standards Critical systems are found in domains such as railway, healthcare and medicine, robotics, aerospace or automotive. Each domain has different regulatory bodies, different organizational and cost structures and different development philosophies; for example, cars are developed differently from airplanes because of the paramount importance of unit cost in the automotive industry. These differences are captured in each domain's safety standards such as DO-178C for aviation, EN50129 for rail, IEC62304 for medical device software or ISO26262 for automotive systems. These standards all reflect the philosophies expressed in the generic safety standard IEC61508.

Tools The standards describe requirements for tools used in the development of critical systems. For the development of critical *software* (as opposed to *systems*), three categories of tools are relevant: development tools create artifacts that execute as part of a CSC (for example, compilers, code generators); analysis tools ensure some aspect of correctness of the CSC (for example, code style checkers, data flow analyzers); and management tools support the development process (for example, managing requirements or test results). Development tools, of which LWBs and DSLs are examples, imply the biggest risk, because they may introduce *additional systematic errors* into the CSC if they are faulty. The standards in all domains require the reliable mitigation of such errors by limiting the permitted tools, introducing redundancy in the CSC, as well as requiring adherence to specific development processes as well as extensive documentation.

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 project-specific 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 exist: (1) Provide proof and/or extensive validation that the tool is correct. (2) The tool itself has been 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 successfully in many similar projects, reports about malfunctioning of the tool have been collected and process-based mitigations (for example, though additional testing) are defined; projects that use a tool from this category must then document that they use the mitigations.

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

2.2 DSLs and Language Workbenches

Domain-Specific Languages DSLs are computer languages optimized for expressing programs in a particular domain [62]. They may have different degrees of expressivity – from simple structure languages to Turing complete languages – and use a wide variety of notations such as text, tables, symbols, math, diagrams [67]. Importantly, the abstractions and notations used in the DSL are aligned with the particular domain at which the DSL is targeted; this is the core difference to general-purpose programming languages, whose abstractions are generic. Many DSLs are used by people who are competent in the domain, but who are not necessarily developers. For those DSLs, notations that reflect the domain are especially important, even if those might not resemble popular syntax from general purpose languages.

Language Workbenches A language workbench is a tool that efficiently supports the development of languages, often, but not necessarily, DSLs; the report from the language workbench challenge [17] provides a good overview. They typically provide a set of DSLs to define various aspects of languages, such as structure, syntax, static semantics, dynamic semantics (usually through interpreters or generators) as well as various aspects relating to IDEs (code completion, syntax coloring, goto definition, find usages, refactorings).

Jetbrains MPS MPS[2] is an open source language workbench developed by Jetbrains over the last 15 years. While not used as widely as some other language workbenches, it is used to implement interesting languages for real-world use. Its distinguishing feature is its projectional editor, which supports practically unlimited language extension and composition [65] as well as a flexible mix of a wide range of textual, tabular, mathematical and graphical notations [67]. The experience of using MPS for implementing a large set of C extensions in the context of mbeddr [71] is discussed in [73]; the paper provides a detailed assessment of the strengths and limitations of MPS for large-scale language development.

Model vs. Program The two terms are used differently in programming language and modeling communities. In particular, there is sometimes a distinction between the general notion of a model and an *executable* model. A program can then be seen as an executable model. Models can also be regarded as a system's description that is more abstract (i.e., contains fewer details) than a program; a program "implements" the model. In this paper, we avoid the philosophical differences and instead use the following pragmatic definitions: every artifact that is expressed with a language defined in a language workbench is called a model. Artifacts expressed outside the LWB, i.e., typically some generated or manually written code expressed in a general-purpose language, we call a program.

Execution Engine The models created with a DSL can be executed either by code generation to a lower level program-

[2] http://jetbrains.com/mps

ming language (which is then in turn compiled) or through interpretation where the program is traversed and the semantics of each syntax element is applied (this may happen either directly on the program syntax tree or on an intermediate represenation that is derived from the program). Both approaches have their own benefits and drawbacks, mostly regarding performance and deployment effort; however, in many context the two are interchangeable. We use the term execution engine to refer to both generation/compilation/execution and interpretation with potential generation of an intermediate representation.

2.3 Verification and Validation

Verification vs. Validation *Verification* ensures that the software works without intrinsic faults. The to-be-verified properties are either implied by the programming or modeling formalism (such as uninitialized reads, invalid dereferencing of pointers or unreachable states) or explicitly specified through test cases or property specifications (postconditions for functions or specific temporal logic properties for state machines). 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 of implementation artifacts to requirements. Not all of these can be automated and some of them are performed by stakeholders other than developers. There is consideral variability in how these terms are defined in various domains (for example, the FDA has a definition for medical devices[3]), but they all fit roughly with the distinction defined here.

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

Coverage Both testing and static analysis suffer from the coverage problem: a fault is only detected if the engineer writes a test case or specifies a verification property that is able to detect that fault. As a remedy, code reviews may find that some tests/properties are missing, various kinds of coverage [77] may be measured by tools, or test generation [10] may automatically raise coverage to a required level.

3. The Benefits of DSLs and LWBs

Our argument 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 backing in this section, we assume that the reader accepts this claim, based on their own experience and the extensively documented productivity benefits of modeling, DSLs, code generation and tools [6, 7, 25, 29, 30, 32, 43, 44, 72]. We include this chapter in this paper even though it is not a contribution, because we refer to the benefits from the case study in Section 6, which also confirms many of these benefits.

Implementation Effort A DSL can reduce the implemenation effort as a consequence of its more appropriate, higher level abstractions. DSL code is typically more concise and requires less boiler-plate than functionally equivalent programming language code [72]. Targeting multiple platforms amplifies this benefit. Low-level mistakes (for example, faulty pointer arithmetics in C) are prevented. IDE support can also be better because the domain-related semantics of the DSL can be known by the IDE.

Note that in critical software, the implementation is not where most of the effort is spent; instead, it is spent in validation and verification (for example, Voluntis calculates three times more effort for validation and verification than for implementation). However, taken together with the validation and verification advantages outlined below, a more efficient implementation allows 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, for example, `switch`-statements in C, an automated analysis to detect dead or unreachable states is much simpler to perform and hence, to implement [74]. Another example is the use of decision tables instead of nested `if` statements. The semantics of the decision table imply that it has to be *complete* (all combinations of inputs must be covered) and *overlap-free* (for every input, only one branch is valid). A solver can be used to check for these properties [56, 74]. A structure composed from `if` statements cannot be assumed to imply these semantics.

Verification properties or test cases can also be expressed at the higher level of abstraction, thus making verification more efficient. For example, 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 [49], even though the lifting of low-level verification results back to the domain level can be non-trivial (there is a whole section (4.1) on testing lifting algorithms in [55]). Clearer semantics are also useful for test case generation. For example, variables can be annotated with ranges or other constraints; the test case generator can use those as the boundaries for the tests (instead of the system-specific, and domain-irrelevant, `MAX_INT` and `MIN_INT` boundaries).

Validation Models can be used to front load [47] validation, reducing the cost of errors [5]. Models can be simulated and tested to ensure that they behave correctly; this approach

[3] `https://www.fda.gov/MedicalDevices/ucm085281.htm#_Toc517237938`

is called model-in-the-loop testing (in systems engineering) or quality-by-design (in the pharmaceutical industry). Validation also involves reviews by other developers or by a separate QA team. Models that use appropriate abstractions and notations make reviews more efficient because they are easier to comprehend and easier to relate to requirements because the semantic gap is narrower; Kosar et al. confirm empirically that program comprehension is improved with DSLs [35]. For validation at the model level to work, the semantics of the DSL must be clear to everybody involved: we briefly address this at the beginning of Section 5.4. In general, non-programmer stakeholders (systems engineers, healthcare professionals, space scientists) can be integrated earlier and more efficiently. Note that even if they do not validate the models directly by inspection or review, the fact that development becomes more efficient and models can be simulated before the implementation is finished shortens iteration times, thereby making the overall process faster.

Finally, tracing of design, implementation and test artifacts to requirements can be more easily supported [70] on models than on code. Models typically exhibit higher locality of features, they are less "distributed" over the model because of the closer alignment of the model with the domain; fewer trace links are required. Fine-grained tracing has been identified as a major problem with current modeling or programming tools [37].

Derivation of Artifacts In critical domains, a substantial number of documents are required as evidence of the correct functioning of a CSC, to demonstrate the adherence to the development process or to make the system's behaviors understandable for non-programmer stakeholders and reviewers. When using DSLs, even though some of the documents may not strictly be necessary because the models are aligned with the domain better, reviewers or certification authorities may still require them. Generating these documents from models (to the degree possible) ensures consistency with the actual system and further reduces effort [80]. Examples include diagrams representing the structure or behavior of the system as well as trace reports.

4. Motivation and Problem

A LWB can be used to define DSLs optimized for the application domain of the CSC. The models created with the DSL are then used to describe one or more particular CSCs.

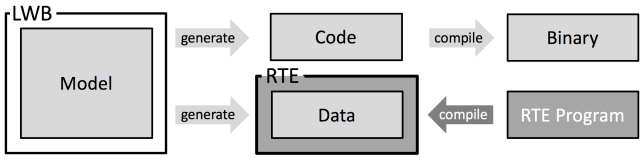

Figure 1. From the model we generate source code, which is then compiled to a binary and executed. Alternatively, we generate data that is interpreted and executed by a runtime environment (RTE), which is built from its own source code.

The implementation of the CSC is automatically derived from the model (see Figure 1). The implementation is either source code that is then compiled, or data (for example, XML) that is interpreted by an interpeter in a runtime environment (RTE). If the transformation to the executable code is correct, this leads to significant gains in productivity as we have discussed in the previous section. However:

> The approach can introduce additional failure modes because of a faulty execution engine. In the case of generation, errors may lie in the language workbenches' generation framework or in the DSL-specific generators. In the case of interpretation, the interpreter might contain bugs or the generation of the intermediate format might be faulty.

Correctness of the tool can be assumed if the tool is qualified (see Figure 2). However, LWBs and the DSLs developed with them usually cannot be argued to be qualified tools as defined above: (1) both the LWB itself and the definition of practically-sized DSLs in these LWBs are too complex to be formally verified or proven correct in industrial practice, (2) existing LWBs have not been developed using a safety process (a particular DSL could be, but that is only of limited use if the underlying LWB is not), and (3) LWBs are still niche tools and not widely used, our DSLs are often specific to a project; a proven-in-use argument is hard to justify.[4] So:

> How can a non-qualified LWB and custom-built DSLs be used in the development of critical systems? How can we ensure that they do not introduce faults in the CSC?

In effect, we are left with ensuring end-to-end correctness, from the model to the implementation; we treat the combination of model + DSL + generator + LWB as one "untrusted" black box. Due to the potentially high effort involved, this runs the risk that many of the aforementioned economic benefits are voided and it may thus be of limited value. The challenge we address in this paper thus becomes:

> How can a non-qualified LWB and custom-built DSLs be used in the development of critical systems, ensuring that the approach does not introduce faults into the CSC and continuing to exploit the benefits afforded by DSLs?

We now describe an approach to solving this challenge, and validate it with an industry project described in Section 6.

5. Assuring the Correctness of the Code

Execution of the CSC happens on the implementation (generated code or interpreted in an RTE, see Figure 4), so, ultimately, the implementation must be correct. We assume that the model has been validated to be correct through the processes outlined in the previous section, but due to the abstraction gap between models and the implementation (there

[4] There are DSLs that are widely used in a particular domain over years such as Cryptol [42]. In such cases, a proven-in-use argument might be feasible.

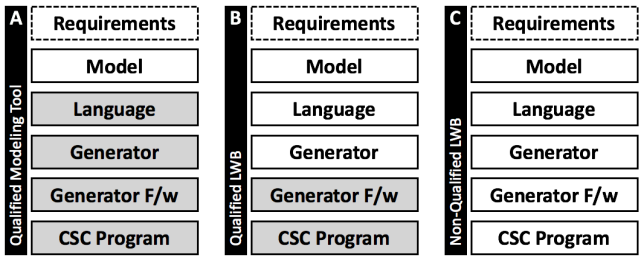

Figure 2. Correctness assumptions (shaded boxes) for different tool chains. (A) When using a qualified modeling tool with a fixed language, only the requirements and the model must be validated. (B) In a hypothetical qualified LWB, one could rely on the generators' transformation from model to code, but the language itself would have to be tested. (C) In the real world of non-qualified LWBs, there is no a-priori trust for any of the components.

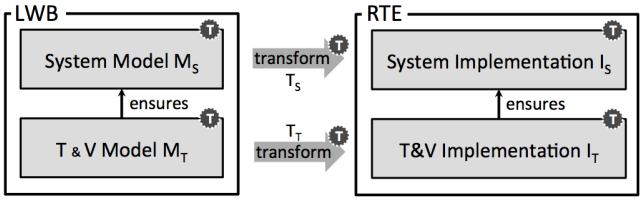

Figure 3. Baseline tool architecture: fundamentally, all tests or analyses (T&V) are expressed at the model level (because this is more convenient and/or productive). However, they are executed at the level of the implementation.

are more details to go wrong at the implementation level), or simply because of bugs in the generator or runtime, the implementation may exhibit behaviours not observed at the model level. So, while we cannot escape the need to perform end-to-end testing (from the model to the implementation, with all intermediate ingredients), our goal is exploit the models to make this as efficient as possible. We rely on the following approach:

> We express both the system and the tests or verification properties on model level and then translate both of them to the implementation and run them there (Figure 3). This way we *express and validate* the semantics on the convenient level of the model (see Section 3), but then *execute and verify* the semantics on the (ultimately relevant) implementation level.

Section 5.1 identifies the risks associated with this approach, and Section 5.2 proposed mititgations; Section 5.3 explains how practices that are also used in a non-DSL-based development process are used in the approach described in this paper. Section 5.4 outlines further safety mechanisms that are not directly related to the use of DSL or the correctness of the generated code, but are still important to overall safety.

The approach works for both tests and static analysis. For testing, the only requirement is that the DSL has a way of expressing test scenarios and assertions. For static analysis,

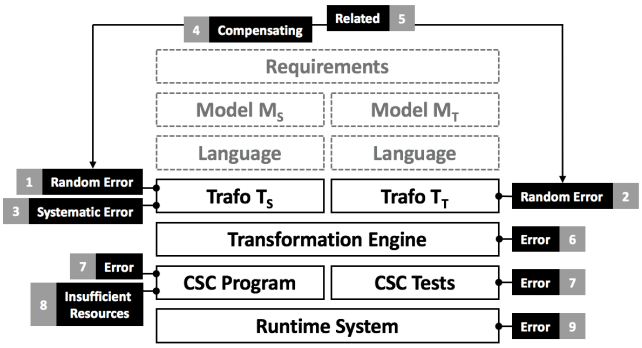

Figure 4. The ingredients of the architecture plus the risks associated with each component.

the DSL also needs a way of expressing the to-be-verified properties on model level. Since properties often come with their own formalism, providing a property specification language may be more effort than just supporting the definition of tests. In addition, the approach only works if an analysis tool for relevant properties is available for the target language.[5] An example is model checking, which can be done by translating 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 [49] and [56]).

5.1 Risk Analysis

We have systematically analysed the components involved in the architecture for possible risks with regards to a faulty CSC implementation; Figure 4 shows an overview. We refer to the numbers in the discussion below. Again, we assume that the requirements and the models have been validated and thus describe the intended behaviors, which is why we do not further discuss the risks of a faulty model here. Validation of the models implicitly also validates the languages in terms of their suitability to express the models; nonetheless we revisit language correctness in Section 5.4. The major risks implied in the language, namely its faulty transformation to executable code, is represented by the two Trafo (short for transformation) components in Figure 4.

5.2 Assuring the Transformation

In this section we introduce mitigations for the risks identified above. They are highlighted in SMALL CAPS in the text below. Figure 5 shows the final architecture with mitigations included. In particular, we ensure that the system M_S the test model and M_T are transformed correctly to the implementation I_S and I_T by transformations T_S and T_T (see the caption of Figure 3 for the meaning of the abbreviations).

[5] Building our own program analysis tools is completely infeasible in practice; it is also not recommended, because those tools must be proven in use (or proven correct) for them to be of any practical use.

RISK 1 **T_S has a random error and generates some system behaviors wrongly.**

Example: When generating a `switch`-based implementation of a state machine in C, the `break` statements are missing, and because of fallthrough, transitions behave wrongly.

Since the tests in M_T assume the correct behavior, and since T_T is correct, the tests in I_T, will fail and detect the error in T_S. We measure coverage on model level to ensure we have enough tests to uncover the problem.

RISK 2 **T_T has a random error and generates some test implementations wrongly.**

Example: Because of an error in the way the editor constructs the model (parsing error, wrong precedence), negative numbers in the model lose their minus sign; they become positive in the generated code (or the intermediate representation).

Since the tests in M_T test the system model M_S correctly, and since T_S is correct, the tests in I_S will fail and detect the error in T_T. Again, we measure coverage on model level to ensure we have enough tests.

RISK 3 **T_T has a systematic error that results in the implementation of all (or some class of) tests to be wrong.**

Example: In template-based generators it is common to first write a syntactically correct, but semantically wrong/trivial version of the to-be-generated code. For example, the developer might initially implement the generator for some model-level assertion as `assert true` and then forget to change the `true` to reflect the actual condition. *All* tests will succeed. An example that leads to a particular *class of tests* to fail is that part of the generator that computes the structure of the code for a particular kind of expression throws an exception. Because the exception is ignored (a bad but common practice), `assert`s that use this expression default to `true`.

These problems 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, but won't, because of the faults in the generators.

> *The implementation of some test cases should be* REVIEW*ed. Alternatively,* FUZZING *(see Figure 5) should be used to prevent systematic errors in test cases.*

Example Resolved: The `assert true` test implementations will succeed, even though fuzzing introduced errors into the code; this reveals the problem. The same is true for the other example.

For some cases, including the `assert true` given above, static analysis of the generator itself can also help. For example, `assert true` could be detected as a constant condition without side effect, which is always wrong/unnecessary. Whether this is feasible depends on whether static analysis for the generation language are available.

RISK 2+3 **Both T_S and T_T have random errors that are not realted to each other.**

Example: 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.

RISK 4 **T_S and T_T have unrelated errors, that coincidentally compensat each other.**

Example: T_S translates actions in hierarchical state machines wrongly in terms of their execution order (entry-transition-exit instead of exit-transition-entry). The generator for M_T translates lists of assertions in reverse order. Thus the assertions assert the wrong order which the T_S generator happens to create.

Unrelated errors, by definition, do *not* result from a (internally consistent) misunderstanding of the requirements and a downstream consistent, but ultimately wrong transformation. Truly unrelated but compensating errors are thus exceedingly unlikely (cf. the contrived example).

> *Implement a second,* REDUNDANT EXECUTION *engine for both M_S and M_T, for example, through an interpreter. All tests must succeed in both cases. It is extremely unlikely for those two to have the same pair of unrelated but compensating errors.*

Example Resolved: The redundant execution engine will not have the list-related error, so the tests will fail there revealing the problem in the transformation for the other execution engine.

RISK 5 **The two transformations T_S and T_T have related (technical or functional) errors.**

Example: Both T_S and T_T transform a model-level `number` type to a C-level `int8`s instead of the required `int16`s, leading to overflow. The tests won't detect it (because both the actual and expected value in the assertion wrap around due to overflow), but the behavior is still semantically wrong.

The relatedness of the two errors in T_T and T_S usually results from the fact that requirements have been misunderstood (functional errors) or that the mapping to the target language is wrongly designed (technical errors). For the errors to be *related* and not randomly compensating, they are most likely the result of the same person or group making wrong decisions (note that errors in the transformation engine itself are discussed in the next Risk paragraph).

> REDUNDANT EXECUTION *reveals the technical errors because different execution platforms will not require the same decisions regarding the mapping to the execution platform. Functional errors should be revealed through the validation process. To reduce the likelihood of related errors in the first place, T_S and T_T should be implemented by* DIFFERENT DEVELOPERS.

Example Resolved: The redundant execution in the interpreter uses Java's `BigDecimal` which does not overflow and/or wrap-around; the execution of the tests in the interpreter will thus diverge from the execution in the generated code.

RISK 6 The transformation engine itself has an error.

Example: Polymorphic dispatch in transformation rules is faulty, applying the wrong transformation rules in some cases.

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

> *If you do not trust the transformation engine, make sure that the* REDUNDANT EXECUTION *does not use that same transformation engine. Diverging tests will reveal the problems.*

Example Resolved: The redundant execution in an interpreter does not rely on the transformation engine and its faulty dispatch; diverging test failures will reveal the problem.

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 use of DSLs.

RISK 7 Low-level failures because of the specifics of the implementation code or language.

Example: Stack overflows, numeric precision errors, timing violations, or invalid pointer derefencings.

To prevent low-level failures as a consequence of specifics of the execution platform, make sure that all (relevant) code paths of the implementation are executed.

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

If REDUNDANT EXECUTION on engines E_1 and E_2 is used, and coverage is easier to measure on E_1 (for example, in the interpreter) you can also "transfer" the coverage to E_2: if the coverage of tests on E_1 is sufficient (i.e., close to 100%), then running the same set of tests on E_2 implies a similar coverage of the relevant code parts there, because the tests exercise the complete, relevant implementation, whatever its specific structure may be.[6]

For some languages, static analysis tools that prove the absence of some classes of errors are available (such as Astrée [13] or Polyspace for C [50]). At least for typical errors, their use is straightforward and is recommended. The code can be generated to use patterns that simplify the analysis, and semantic annotations can be added that enable more meaningful analysis (for example, for analyses based on Frama-C Jessie [14]). The semantic information is available in M_S and can be mapped by T_S. An example that verifies concurrent C programs is given in [15].

> *Use* STATIC ANALYSIS *tools to further increase the quality and reliability of the generated code; generate analyzable patterns via T_S and add semantic annotations required by the code-level analyzer based on M_S.*

RISK 7 The implementation may be exploited maliciously as part of an attack on the system.

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

Because of the degrees of freedom in the implementation, the system may be attacked by exploiting those degrees of freedom. Those attacks can usually not be predicted from the model level, because the model (intentionally) abstracts from those degrees of freedom. Penetration testing [2] on implementation level can help prevent those.

> *Perform* PENETRATION TESTING *on I_S to ensure the absence of attack vectors.*

The generator can generate potentially more secure code [68], for example by calling sanitizing functions for all inputs [53]. Automating this through a generator avoids relying on developers' consistency when doing it manually.

RISK 8 Insufficient resources may lead to errors.

Example: A data queue associated with a sensor overflows because of an unexpectedly high signal rate on the sensor; data is lost, the system behaves wrongly.

In contrast to a testing or simulation environment, 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. Make those limits explicit in the documentation.*

Capturing the expected resource utilization explicitly in M_S and then monitoring it during execution helps with diagnostics because meaningful error messages can be issued – resource starvation is hard to debug otherwise.

RISK 9 The target compiler or runtime system may have errors.

Example: The interpreter handles operator precedence wrongly.

When using interpretation to execute the models, handling this risk is mandatory, and conceptually related to the previously discussed case of faulty transformations. If a generative approach with a downstream proven-in-use compiler is used, it is very unlikely that the compiler has errors and one can probably avoid this step.

[6] Note that there might be *additional* code/behaviors in E_2 that could be exploited maliciously. We discuss this below.

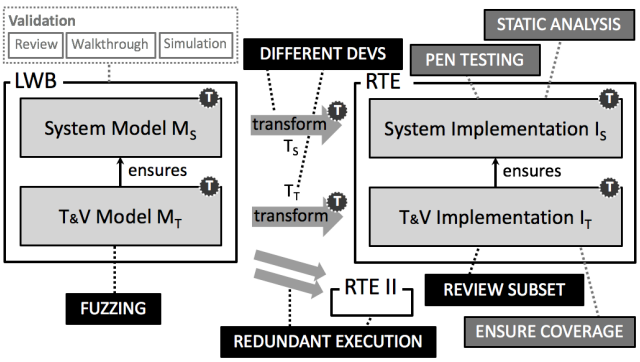

Figure 5. Annotated baseline architecture from Figure 3. Black boxes represent code verification activities that are necessary because LWBs and code generation are used; grey boxes are verification activities that would be done in the same way if the code was written manually; and white boxes are activities to validate the quality of the model. The circled Ts represent traces to requirements.

> *Execute the system in* REDUNDANT EXECUTION *engines to find problems in either of them; for the generative approach, this may involve the use of different compilers.*

Essentially, one can treat the transformation/compilation/runtime stack as a black box and use redundancy and test divergence to find errors in either of them.

5.4 Additional Safety Mechanisms

In addition to the risks identified in this paper, other things can go wrong in critical systems; many of those mentioned mentioned by Koopman in [34] are relevant. In this section we discuss a few of them that are especially relevant.

Well-defined Language Users of the DSLs have to understand its semantics to be able to create correct M_S and M_T models, and to validate those. The same is true for the developers of T_S and T_T so they can "implement" these semantics in I_S and I_T. To this end, a clear definition of the language and its semantics is needed. For the end user, the tests in M_T or the ability to simulate the execution of the models can help with understanding the semantics.

QA'ing the Language This paper presupposes that we cannot assure the correctness of the DSLs and the transformations themselves (although there is work on this [1, 45, 63]), which is why we propose an architecture that remedies resulting risks. Nonetheless, the DSLs *should* be systematically tested and verified to the degree this is practical. Work on language testing includes execution semantics [81], static semantics [18], grammars and concrete syntax [39] as well as multi-aspect testing frameworks [31, 55]. Ratiu and Voelter specifically discusses language testing in MPS [55], using manually written type system tests, automatically generated test cases for language structure and syntax, and measuring transformation coverage.

Quality of Generated Code Some domains require code to conform to particular guidelines. For example, C code in automotive systems is expected to conform to MISRA-C [48] 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, for example the one by LDRA.[7] Generating code that conforms to such guidelines is easy if the transformation developer is aware of them. The strictness of the guidelines may be reduced for generated code. For example, customers have granted us MISRA exceptions for generated polymorphic dispatch code that uses (otherwise prohibited) void pointers.

Architectural Mechanisms In this paper, we discuss how measures in the development process and tools can assure CSC correctness. An orthogonal approach is runtime monitoring and fault detection [22], which transitions the system into a safe state if a fault is detected. In medical systems, a safe state may be to pop up a message that tells the user to disregard the software and call her doctor. Examples of architectural mechanisms for runtime monitoring 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 [20]).

5.5 Summary of the Approach

Figure 5 summarizes the overall approach: we express both the system and the tests or verification properties on model level and then translate both of them to the implementation and run them there. This way we *express and validate* the semantics on the convenient level of the model, but then *execute and verify* the semantics on the implementation level. To address the remaining risks, we

- *use redundant execution on two execution engines,*
- *use different developers for the two transformations,*
- *review a subset of the generated code,*
- *clearly define and QA the DSL,*
- to use fuzzing on the tests,
- ensure high coverage for the tests,
- run the tests on the final device,
- perform static analysis on the generated code,
- perform penetration testing on the final system,
- and use architectural safety mechanisms.

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

Only the first four, those printed in italics, are specific to our use of DSLs and LWBs; the others would be performed in any case. We now evaluate this approach with a case study in the healthcare domain, emphasising how the use of DSLs impacts these activities and their economic feasibility.

6. Case Study in the Healthcare Domain

In this section we validate our approach with a case study from the healthcare domain. In particular, we discuss the DSL-based development of software medical devices at Voluntis as part of the PLUTO project. The development process has to conform to the requirements of the Food and Drug Agency (FDA) for the device to be used in practice.

6.1 FDA Requirements on Medical Software

Selling software in a medical context in the US requires authorization from the FDA.[8] The FDA defines the notion of a Software as a Medical Device (SaMD), which is a software-only solution for diagnosing or treating diseases.[9] In the relevant FDA documents[10] an SaMD is classified by Level of Concern (LOC): Minor, Moderate, and Major. Both the SaMD and any off-the-shelf software that is used in the SaMD require a hazard analysis, basic documentation, hazard mitigation, and a description and justification of residual risk. If after hazard mitigation the LOC is still High, they require so-called special documentation.

Basic documentation encompasses a description of what the software is, the hardware it requires, how the end user is guided (to help avoid risks that result from wrong usage), 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 mitigates[11] these hazards, including protective measures, user warnings, user manuals or special labelling materials. The required *justification of residual* risk then usually does not contain any remaining ignificant risks.[12]

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. For DSL-based development, this includes systematic management of the requirements for the DSLs, tracing of requirements to the language definition, as well as the well-definedness of the language and sufficient testing (see *Defined Language* and *Language QA* in Section 5.4).

The FDA has found that the majority of software-related device failures are due to design errors:[13] the most common problem was failure to validate software prior to routine use. DSLs can help with this; see Section 3. As discussed in Section 5, LWBs and DSLs introduce additional hazards, and we also show in Section 5 how these are mitigated in principle. Voluntis are confident that the benefits of introducing DSLs and LWBs will outweigh their risks and thus produce a beneficial risk-benefit ratio, accelerating the production scale-up and reducing the cost of QA of medical device software. The goal of this case study is to prove this point.

Since the documentation requirements are essentially similar to other high risk software components, we do not discuss the details in this case study. In the remainder of this chapter we illustrate how we have implemented the technical means of ensuring the quality of the CSC in the context of the Voluntis PLUTO project.

6.2 Business Context

Because of the safety implications discussed in this paper, the development of SaMDs is expensive and time-consuming. The well-known fact that fixing errors becomes more expensive the later they are found during development [51, 52] is exacerbated in the healthcare domain because much of the test, documentation and review processes required by the standards has to be repeated.

Voluntis' SaMDs are used to help healthcare professionals (HCPs) manage and treat diseases, to calculate the dosage of medication, and manage side-effects of oncology therapies by providing alerts to HCP and medical recommendations to patients. They are realized as web pages and mobile apps, both for Android and iOS. The underlying algorithms are inherently complicated, and their safety and effectiveness must be ensured. For the reasons given above, it is crucial for Voluntis's business success to establish an SaMD algorithm development process that reveals errors in medical algorithms early, and not only when they are deployed on the mobile device for prototype use, or even when they are in the hands of users.

Voluntis has decided on a DSL-based approach that uses models for defining and validating the algorithms in order to benefit from the early validation and simplified review with HCPs discussed in Section 3. The abstraction of the core behaviors also helps avoid duplicate implementation effort for Android and iOS. On the flip side, this requires verifying the correctness of the CSC derived from the models.

[8] Other jurisdictions have other regulating bodies. But the FDA is generally considered to be the most stringent one, so it is commonly used as the benchmark.

[9] `http://www.imdrf.org/docs/imdrf/final/technical/imdrf-tech-131209-samd-key-definitions-140901.pdf`

[10] `https://www.fda.gov/downloads/medicaldevices/deviceregulationandguidance/guidancedocuments/ucm073779.pdf` `https://www.fda.gov/downloads/medicaldevices/deviceregulationandguidance/guidancedocuments/ucm524904.pdf`

[11] The notion is to mitigate them to As Low As Reasonably Possible (ALARP).

[12] In some cases, some High risks could remain, but then it is up to the manufacturer to document that the risk/benefit ratio is better than the already existing solutions. This would still be accepted by the FDA since there is still a benefit.

[13] `https://www.fda.gov/downloads/AboutFDA/CentersOffices/OfficeofMedicalProductsandTobacco/CDRH/CDRHTransparency/UCM388442.pdf`

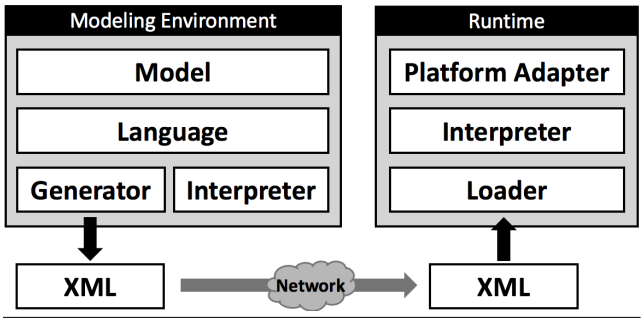

Figure 6. Overview of the overall tool and system architecture; see the running text in Section 6.3 for details.

6.3 Overview of the System and Tools

Figure 6 shows an overview over the system. A *modeling environment* is used by mixed teams of medical/technical professionals to design the algorithms underlying the SaMD. They rely on a set of DSLs built specifically for PLUTO which provide abstractions and notations meaningful to HCPs (Figure 7). These include decision tables, decision trees and numbers with ranges, but also state machines to capture the asynchronous, event-based, time-sensitive execution of the algorithms. Additional languages support is available for defining tests and simulation scenarios as well as for specifying various configuration options for visualizations and simulations. The DSLs are described in some more detail in Appendix A.

At the core is a simple functional expression language called KernelF. KernelF[14] has not been developed specifically for PLUTO; it is an embeddable functional language developed by itemis that has been used in several DSLs in various domains. Of KernelF's 260 concepts, only 83 are availabe in PLUTO; the removed[15] concepts have either been deemed unnecessary in the domain (for example, option types) or have been replaced with alternatives that more closely resemble the medical domain (decision trees and tables). Appendix A provides a few more details about the PLUTO languages and also contains example code.

The sizes of the PLUTO languages are shown in Table 1. To put the size and the associated challenge of ensuring correctness into perspective, we compare to mbeddr [69], a set of C extensions for embedded software development implemented with the same LWB technology as PLUTO. mbeddr has ca. 1,600 language constructs [73], and its correctness has been ensured, mainly through testing, to the point where it was successfully used for commercial software development [72]. The PLUTO languages are roughly one sixth in size, which makes ensuring correctness considerable easier than in mbeddr. In addition, about one third of the PLUTO languages is reused from KernelF, further reducing effort and providing a solid foundation in terms of correctness.

The DSLs are built with MPS, and the modeling environment is a stripped-down version of MPS as well. The environment comes with an *interpreter* that is able to run the algorithms for simulation and testing directly in the modeling environment. The functional core reuses the existing interpreter for KernelF.

Because of deployment constraints (in particular, the time it takes Apple to review apps on the app store), the execution of the algorithms on mobile phones does not rely on generated binary code. Instead, the *runtime environment* embedded in the mobile applications contains a (second, different) interpreter written in C++ which, after separate compilation, runs both on Android and iOS. The interpreter consumes the algorithms in the form of XML files created by a *generator* in the IDE and shipped over the network to the user' devices. Platform-specific libraries and frameworks (whose verification and validation is outside the scope of this paper) are used for integration with the two platforms, for example, for accessing sensors and system values, storing data and UI.

At the core of the challenge thus was ensuring the correctness of the XML generator as well as the interpreter in the runtime; we focus on those below. In addition, the correct transfer of the XML to the devices had to be cryptographically ensured, and the algorithm had to be versioned to ensure reproducibility of an algorithm's execution for a given patient. We do not cover these aspects in this paper.

6.4 Current State

Version one of the system is implemented and tested. Voluntis is planning to release their next-generation products on PLUTO in late 2018. It is a major business concern for Voluntis to obtain the FDA clearances for their software medical devices promptly. Therefore, a team of regulatory experts meticulously challenges and prepares the technical files to ensure they are fit for FDA clearance before submitting them. As of April 2018 the clearance for a new SaMD based on PLUTO has passed the pre-submission phase without any concerns from the FDA regarding the proposed approach.

Going into detail about the intricacies of the clearance would be a different paper; we just provide a very brief outline. FDA clears the complete SaMD, end-to-end. We treat

Language Part	# of concepts	percentage of total
Expressions (KernelF)	83	31%
Expressions (Extended)	63	23%
State Machines	29	11%
Testing, Scenarios	41	15%
Configuration	54	20%
Total	270	100%

Table 1. Size breakdown of the PLUTO languages. One third of the overall language was reused from KernelF.

[14] `http://voelter.de/data/pub/kernelf-reference.pdf`

[15] In addition to extension, the ability to remove language concepts that are not needed as part of a specific DSL is an important ingredient to making an embeddable language truly reusable.

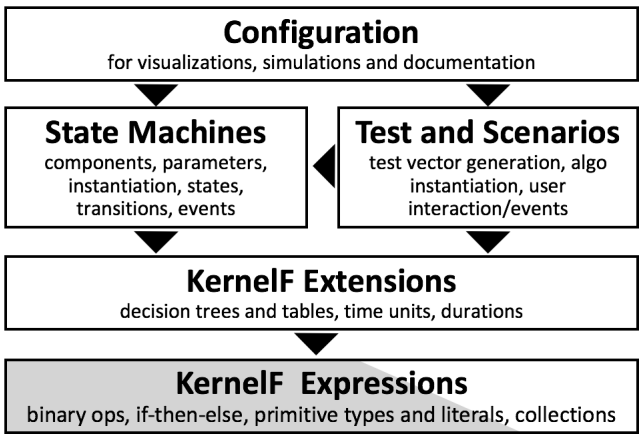

Figure 7. Overview of the various parts of the languages used to implement SaMDs. Arrows represent dependencies; shading indicates that only ca. 1/3 of KernelF was used in this language.

the SaMD developed using the DSL like any other, manually created SaMD. This means that we have to demonstrate and document that that our SaMD has been conceived, verified and validated as extensively as possible. As we describe below, we have achieved test coverage of 100% with regards to the medical algorithms, so we expect no problems here. The reusable runtime environment, including the interpreter, will be treated as off-the-shelf software (OTSS). OTSS has its own requirements regarding safety. In the end it also boils down to demonstrating test coverage. This includes the manual tests for the framework parts, but also the interpreter tests (which is why 100% coverage for the interpreter are useful).

Later in 2018 the DSLs will evolve further to support additional language constructs necessary for additional categories of algorithms. While the algorithms so far have emphasized timing and process, future algorithms will be more sophisticated in terms of querying and filtering of data and computation of derived values. We are also evaluating the integration of analysis tools into the IDE. For example, an SMT solver[16] can be used to statically check the completeness and overlap-freedom of hierarchical boolean expressions (those found in decision trees and tables). The solver integration is already available for KernelF [74] and can be extended to the Voluntis DSLs straightforwardly. In addition, a model checker could be used to verify the absence of dead-end or unreachable states in the state machine.

6.5 Development Tool Architecture

The architecture in Figure 5 is designed to address the risks described in Section 5.2 and Section 5.3. In this section we describe how this architecture was implemented for PLUTO.

Validation As suggested in Section 3, validation of the algorithm is simplified by using the DSL. The decision trees and decision tables are suitable for review directly by HCPs. The state machine requires a degree of fluency in computational thinking [79] that not all HCPs have, which is why a linearized visualization of the algorithm is generated for review. At the core of the validation is the ability to execute the algorithms in the IDE based on the in-IDE interpreter. A simulator makes use of this interpreter to let HCPs simulate the behavior of the models, resembling the interaction with the algorithm on the patient's phone. Particular executions of the simulator can be recorded into test cases for automatic re-execution and detection of regression errors.

One particular example for the usefulness of the interpreter is the following. Bluejay is a blood pressure algorithm. We ran simulations with HCPs from key stakeholders (among them oncologists from a renowned US partner oncology hospital, and a pharma partner). In one case, the scenario covered a patient who had her blood pressure go up quickly over 3 days and had to go to the emergency room. We reran the simulation with the same values entered by the patient over the 3 days, moving forward in time with the simulator. This confirmed that the algorithm behaved as expected and that the patient did not follow the instructions (which led to the emergency room). The simulation took less than five minutes in a conferene call with the doctors.

Redundant Execution The full algorithm behavior and the tests are executed redundantly with the in-IDE interpreter and with the C++-based interpreter in the runtime environment. For the functional core, the existing interpreter for KernelF was reused in the IDE, and a similar interpreter was implemented in C++ for the runtime environment. Since KernelF is purely functional, these interpreters are relatively simple; they can be seen as recursive calls of an `eval` method on the AST nodes. Note that the idea of integrating the C++ interpreter into the IDE (in order to have only one interpreter) was explicitly discarded, because the redundancy in the implementation of the language is advantageous in the context of safety-critical applications. The higher effort for developing two interpreters was accepted. The semantics of the two interpreters were aligned through test cases (we discuss coverage below). Because the Java-based interpreter is much simpler[17] than the one implemented in C++, the Java interpreter was considered the "specification", and in the case of semantic differences, the bug was primarily suspected in the C++ version (an assumption that was correct in most but not all cases). Decision trees and decision tables fit that same functional execution paradigm, and the KernelF interpreter was modularly extended. For the state machine, whose execution model goes beyond pure functional evaluation, two new interpreters were written to deal with state, time and asynchronicity. Here it was a little

[16] `https://en.wikipedia.org/wiki/Satisfiability_modulo_theories`

[17] Many reasons contribute to this: it does not have to care about non-functional concerns, so no optimizations are involved; MPS offers convienient APIs to traverse trees; Java in general requires attention to fewer details than C++, for example as a consequence of garbage collection; and a part of the interpreter could be reused from KernelF.

bit more challenging to align the semantics of the two, but through a sufficient number of representative test cases this was achieved as well.

Note that the single C++ interpreter is compiled to iOS, Android and x86/Linux. This allows us to run all scenarios on three different platforms, automatically, on the build server, which further increases redundancy and reliability.

Testing and Coverage A good test suite serves two purposes. From the perspective of the end user, who assumes that the execution engine is correct, tests verify the behavior of a program. From the perspective of the language developer, who assumes that the programs and the tests are correct, they serve to verify the execution infrastructure. While, in the future, the testing aspects of the DSLs will be used for the former, during the development of the development tools they were used for the latter. For this to be meaningful, a very high coverage relative to the execution engine must be achieved. We measured the following aspects of coverage[18]:

- We started with measuring the use of language concepts and relationships. We achieved 100% coverage here. The coverage analyzer also analyzed the complexity of the part of the model where the concept was used to ensure that the language concepts were not just used in isolation.
- Next, we analyzed the coverage of the in-IDE interpreter: all evaluators for all language concepts had to be visited at least once, and for interpreters that covered different cases (for example, interpreting an empty list vs. a non-empty list), all cases had to be executed at least once. 100% coverage was achieved as well.
- Since we executed all test cases in both interpreters, a given coverage for the in-IDE interpreter automatically translated to a similar coverage of the C++ runtime interpreter. However, as an additional means of confirmation, we also measured the coverage of the C++ implementation of the interpreter using gcov.[19] Finally, we ran the cppcheck[20] analyzer on the C++ implementation to ensure the code quality of the interpreter implementation; no warnings are reported.

The generator that creates the XML from the models in the IDE is generic with regards to the language structure. We tested it using unit tests. We did not measure coverage there specifically, because every model that was transferred to the runtime environment for test execution also implicitly tested the XML generator: a failing test on the runtime might result from a faulty XML generator. We did not encounter any signficant flaws in the XML generator.

[18] Of course, as is always the case with coverage measurements, high coverage is not a guarantee for the absense of errors; for example, one cannot exhaustively test the ranges of (number) values or cases where a language structure allows for an unbounded set of programs.

[19] `https://en.wikipedia.org/wiki/Gcov`

[20] `https://en.wikipedia.org/wiki/Cppcheck`

Earlier we had argued that we cannot qualify the language workbench, the language or the generators. Instead we perform end-to-end testing of the modeled artifacts, automated to the degree possible, with high coverage. This is also the level of proof we provide to FDA.[21] However, as we describe here, we also tested the language itself by aiming for 100% coverage of the interpreter and concept instantiation. This is for two reasons. (1) It helps build our own trust in the system. We are aware of the risks the approach incurs, and we want to do everything that is realistically possible to avoid faults. (2) Testing the infrastructure itself indirectly helps with testing artifacts: if users write tests for their algorithms and a test fails, we want to make sure the problem is actually in their test, and not in the underlying infrastructure.

Fuzzing To ensure that the tests were actually sensitive to the algorithms (and did not just contain `assert true` statements), we used a combination of code reviews and manual fuzzing, i.e., we manually changed some of the algorithms and ensured that the corresponding tests failed.

6.6 Evaluation of the Approach

Test Algorithms Validation and testing has focused on two SaMDs codenamed BlueJay and GreenJay. For the former, existing blood pressure and diarrhea algorithms currently used in clinical studies[22] have been ported to PLUTO. For the latter, four additional side-effect management algorithms have been implemented with PLUTO.

Size of the Test Suites and Coverage To validate Bluejay, 305 test scenarios have been implemented; 28 for diarrhea, 277 for blood pressure. The GreenJay test suite has 296 scenarios (diarrhea 75, fever 109, nausea/vomiting 60, pain 52). Many lower level tests of functional logic used by the algorithms (such as decision tables or helper functions) are available for all algorithms. In Bluejay, 123 test scenarios were adopted from the test specification for the original (manually developed) algorithm. With PLUTO, the algorithm coverage measurement allowed us to improve coverage to 100% of the algorithm. The GreenJay tests provide 100% coverage of the algorithm and the associated functions.

Validation Efforts Compared to the earlier manual implementation, the effort for writing the test cases for Bluejay was reduced from 50 person days (PDs) to 15 PDs. This 70% reduction was attributed to the fact that they could be written on the level of the DSL as opposed to the implementation. For GreenJay's four algorithms it took 55 PDs; no comparison is available because the GreenJay algorithms were new.

Even in the original approach, a high test coverage of the algorithms was attempted. For this, test vectors were systematically created based on equivalence classes [28]. The

[21] In conversations with people from FDA we have learned that static analysis will play an increasing role in their assessment of the quality of a software system. However, currently, testing and documentation is still paramount.

[22] `https://clinicaltrials.gov/ct2/show/NCT02345265`

verification engineer defined all the possible inputs and corresponding outputs for each medically relevant path in the algorithm in an Excel sheet, then manually reduced the data set by removing redundant input values (those that lead to the same output by going through the same logical path in the algorithms), and then write a test for each remaining input. In the new system a generator performs the generations and simplifications automatically; the team estimates a factor of 20 in reduction of the effort.

The unavoidable change requests are also handled more efficiently. The original process involved several manual steps, synchronized manually over several tools: (1) write requirements specs in Excel incl. flow charts (HCP); (2) implement software (dev); (3) write test cases (tester); (4) build app (dev) and test it (tester). With PLUTO, everything is done in the IDE, jointly by HCPs and developers. Various consistency checks are applied, and execution happens directly in the IDE. This shortenes the iteration time between HCPs and developers significantly (and reduces cost).

A concrete example: several rounds of refinement of a blood pressure algorithm were performed with the same stakeholders mentioned above (oncologists from a US partner hospital, and a pharma company). The refinement objective was to allow HCPs to define thresholds to trigger medical recommendations to patients to call their medical team. Using PLUTO, the team was able to define the requirements in week one and then implement the change and run the simulation on real data with the partners in week two. Then they requested some changes, which were implemented the next day, and in week 3 the simulation was demonstrated and the partners validated the change. With PLUTO, the process could easily have happened in one week (it took three because of the limited availability of the hospital's HCPs). This is much faster compared to pre-PLUTO interactions.

For verification and validation the team also reports "tremendous" speedups: if a change is necessary after verification/validation, large parts of the verification/validation have to be repeated. In the old approach, for each (group of) change(s), testers had to go through all existing tests and adapt them, then reexecute all. What, for Bluejay, could take up to 10 PDs with the old way, is now done in 10 minutes. Considering that sometimes up to 5 change/verify/validate rounds are necessary, the overall improvements are indeed "tremendous". The need to support both iOS and Android adds to the gains in efficiency. During initial algorithm development only one platform would be supported to avoid duplicate effort. But for changes after clearance, both platforms have to be re-implemented/tested. The ability to validate the algorithm on model level and the automated execution and testing on the two platforms contributes to the reduction in effort from 10 days to 10 minutes.

Tool Development Effort The development effort for PLUTO itself, i.e., the languages, the IDE, the interpreter and simulator, the XML-based transfer to the client as well as the C++-based interpreter in the runtime as well as the platform-specific adapter code was about 1,000 PDs, including testing and validation. This is a significant investment for Voluntis. However, considering the downstream increases in productivity outlined above, as well as the resulting reduction in time-to-market make this a valid business case. Various meetings with the board have confirmed the approach.

Tracing There are two approaches for tracing models/code to requirements. The one mentioned earlier attaches trace links to model elements that point to the requirements that influenced the particular element. Because of Voluntis' use of MPS, the technical approach described in [70] would work, where the trace links are attached directly to model elements in MPS; a prototype has been developed that links directly to specifications, software items and units in Team Foundation Server. For now, Voluntis decided to use the approach they had relied on in the past, mostly to be able to use existing reporting infrastructure. Here, requirements IDs are added into commit messages and existing tools are used to extract the trace reports necessary for FDA clearance.

Use of Additional Safety Mechanisms This section discusses how the mechanisms introduced in Section 5.4 were used in the context of this project.

The language was *well-defined*; it was developed together with domain experts, implemented in a principled way using the LWB, and also documented for the end users. A formal semantics, which would guarantee soundness of the language definition, was not provided; this would require a completely different approach, skills and tooling. Likely it would have resulted in a much more restricted language, which in turn conflicted with the goal of making it accessible to domain experts to facilitate validation.

As we have discussed before, to *assure the quality of the languages*, we have systematically tested the interpreter to 100% coverage. In addition, we have used MPS' test support [55] to test the static semantics (type system) of the languages. The language was also designed to constructively avoid certain classes of faults. For example, numbers with ranges (with some static checking) avoid overflows, and high-level decision expressions such as tables or trees avoid complex `if` cascades. The top level structure of the algorithm's behavior was expressed as state machine because of its well-defined semantics, with the intent of integrating model checking in subsequent versions of the language.

Quality of generated code is not relevant: apart from the generic XML generator, no code generation was used. The conceptual equivalent was the QA of the C++ runtime; we have explained above how we verified this through testing.

We used *architectural mechanisms* to identify failures at runtime. In particular, the runtime system performed periodic consistency checks of critical external data such as the system clock or the elapsed time since the last network connection (for possible updates of the algorithm). The language also supports the definition of global and state-local invari-

ants that are checked by the interpreter. A failure of an invariant would stop the interpreter and go into a safe mode (tell the patient to call support).

Reviews Reviews of models and simulation results are an inherent part of validation; see above. We tried to automate the verification of the runtime system as far as possible with the means discussed above. However, for the parts of the runtime outside of the interpreter (such as the interaction with operating system services) regular implementation code reviews and manually-written unit tests were used.

Different Developers The two different execution engines, the in-IDE Java interpreter and the C++-based interpreter, were developed by different teams. While this was not done explicitly to avoid the risks of implementing both consistently wrongly, but just because of different developer skills (Java vs. C++), the effect was the same: a misunderstanding of the requirements would not automatically lead to consistently wrong behavior of both runtimes.

Pen Testing Penetration testing has not been performed specifically for the interpreter, because it is not exposed to the world. However, the platform-specific runtime system, as well as the final apps, will undergo security testing, including penetration testing. We do not discuss this any further, because it is not specific to the DSL-based approach.

6.7 Lessons Learned

Execution is Crucial The core benefit perceived by the HCPs is the ability to simulate ("play with") an algorithm. Of course an executable model of the algorithm is a necessary prerequisite, so this does not invalidate the approach. But it changed our effort allocation: we put more efforts into a realistic simulator that closely resembled the final app.

Underestimated need for Computational Thinking Decision trees and tables are maintained by HCPs. However, as a consequence of the asynchronicity and the time dependence that is part of the algorithm core, the HCPs had a hard time working directly on that part of the algorithm, even though the DSL features declarative constructs to work with time and durations. As of now, these parts of algorithms are implemented by technical people and reviewed by HCPs. Further training of the HCPs is neccessary. More generally, educating domain experts in the basics of programming and computational thinking [79], is useful in a wide range of domains, not just this particular one. Among other things, we are working on an interactive tutorial[23] that teaches programming novices about values, expressions, dependencies, functions and the like.

6.8 Threats to Validity

Internal Several of the authors were involved in developing PLUTO, which might lead to bias regarding its success. However, our judgement relies on feedback from users and on concrete numbers; we believe the conclusion is valid.

A second caveat is that we base our conclusion on *early* experiences with the approach. It is possible that future algorithms require so much change to the DSLs and runtime that the initial benefits are voided. However, this is very unlikely because the effort for incrementally adding a few language concepts is in the order of a few days, plus a few weeks with verification and valiation. In addition, the Greenjay and Bluejay algorithms have been chosen because they are representative of at least the known set of future algorithms, based on Voluntis' considerable experience in the domain.

To the degree that changes to the languages will be required in the future, our 100% test coverage for the interpreter (and the additional type system tests) ensures that changes that break existing semantics will be revealed. The mechanics of language/model co-evolution are handled by MPS, as discussed in *Evolution of Models*, page 35, of [73].

Another possible threat to internal validity is that we can only have limited experience regarding the willingness of the HCPs to work with PLUTO. The HCPs we have interacted with so far were very positive, but those were of course chosen because they were open to the idea. On the other hand, even if HCPs would never *write* algorithms but only reviewing the models and experiment with the simulators (with the modeling done by developers), almost all of the benefits discussed in Section 6.6 still apply.

External The applicability of the approach in other safety-critical domains, beyond healthcare, is of course crucial for external validity. We dedicate the Section 7 to this topic.

Another aspect to external validity is the applicability of the results outside Voluntis, i.e., other companies in the healthcare domain. As long as the domain expertise and the management support is available, we cannot see why the approach would not work in other, similar settings.

The final question for external validity is the degree to which the approach depends on using MPS as the language workbench. The core approach to establishing safety does not depend on MPS; for example, it could be implemented with Xtext and Xtend instead [4]. However, based on our experience with Xtext, we expect the language development effort to be higher. However, other factors might be more important. For example, the ability to reuse an existing, reliable expression language such as KernelF requires support for modular language composition. It is available in MPS, but Xtext supports only a much more limited form; many other language workbenches do not support it at all. Similarly, MPS' support for non-textual notations is crucial for a language that faciltates efficient validation (for example, the decision tables and decision trees). Xtext does not directly support mixed notations, but they could be built with other Eclipse-based UI technologies. The effort, however, would certainly be significantly higher. Summing up, it is less ob-

[23] `https://markusvoelter.github.io/ProgrammingBasics/`

vious how language workbenches other than MPS could be used to realize this approach, though it is definitely possible.

7. Other Domains

We now provide brief discussions of using the approach for three additional domains: space, automotive, and robotics.

Space/ESA The ESA-founded European Cooperation for Space Standardization[24] (ECSS) is an initiative for developing a coherent, single set of standards for all European space activities, and in particular, flight software. 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. However, a comprehensive prototype based on mbeddr C has been developed by OHB System AG [80]. Initial feedback from ESA is positive, but no formal acceptance has been attempted yet.

Automotive Major trends in automotive software (for example, 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 (for example, AUTOSAR[25] for generating component glue 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-based development tools is addressed by proven-in-use arguments and by treating the code as if it was manually written.

ISO26262 also specifies that, for untrusted tools, there must be "very high confidence" that errors are detected. Tool vendors provide reference workflows (for example, [12]) that define process activities to satisfy this requirement. Most activities defined by these workflows are covered by the LWB approach as described in Section 5, for example, 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 usually cannot be proven-in-use, some additional measures are needed compared to the state-of-the-art reference approach described in [12]. 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 are confident that the approach proposed in this paper will work in automotive software. Two projects are currently being developed for customers of itemis[26] that will shed light on this assumption.

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 [21]. All of these properties fit well with DSLs and LWBs (Section 3).

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 static analysis to reduce the test effort. For those to be adopted more widely across industry, they must 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. The architecture proposed in Section 5 seems suitable to enable easier access to domain expert, and therefore reducing time and cost of adoption, as discussed in Section 4.

[24] `http://www.ecss.nl/`

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

[26] We are not allowed to mention names at this point.

8. Related Work

Overviews A general comparison of safety standards for different domains is presented in [40]. While it does not directly consider DSLs, LWBs or code generation, it provides a good overview over the general challenges. Conmy and Paige [11] focus 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.

DSLs in Safety-Critical Software Tolvanen et al. report about the use of a graphical DSL (implemented in MetaEdit+) in a medical system [61]. They observe some of the same benefits in terms of validation and integration of stakeholders. However, the language deals with lower-level aspects of a controller in a medical device, and the stakeholders are presumably technical (as opposed to our HCPs). While they also generate code, they are silent about if or how they assure the correctness of the generated code.

Florence et al. discusses POP-PL, a DSL for describing complex medical prescriptions [19]. Similar to the PLUTO case study, it expresses a prescription and monitoring process as a reactive system. The paper contains an evaluation of the DSL with real users based on a survey; they report that prescribers and clinicians can understand programs in this language. The paper also contains a formal execution model based on the lambda calculus. However, the paper does not address safety and certification. In other words, it confirms the benefits of DSLs and models introduced in Section 3, but does not address dealing with possible risks (Section 5).

DSLs have been used in other critical domains. For example, Réveillère et al. discuss operating system driver development [58]. Haxthausen and Peleske [24], as well as Svendsen et al. [60] describe DSL for defining railway interlocking protocols. All of these works exploit the benefits of better analyzability on DSL level and identify the *need* for verifying the interpreter/generator and the compiler, but do not *address* this challenge. In contrast to our paper they also do not describe an architecture that compensates for this shortcoming, for example, through systematic testing.

Tools Proven in Use Not surprisingly, the largest stack of related work in the context of generating safety-critical software from models relies on tools that are proven in use. For example, Pajic et al. [54] use UPAAL for modeling and verification of models, then translate the models to Simulink/Stateflow, and finally use its proven-in-use code generator to generate the C implementation. Note that the transformation from UPAAL to Simulink is not proven-in-use; it has been "verified" through reviews. Beine and his coauthors rely on dSpace's TargetLink as well [3]. In general, proven tools will force developers to use existing (proven) notations, which is problematic regarding the benefits of models for validation, because they are usually not very well aligned with a particular domain. Stacking a DSL on top of a proven lower-level modeling tool is useful though, because at least a part of the semantic gap is bridged by proven tools; the remaining gap is smaller, and thus presumably easier to verify. Building DSLs on top of KernelF is a small step in this direction.

Provably Correct Tools To the best of our knowledge, the only qualified modeling tool that is proven correct, and that is used in industrial practice is Scade [16]. However, similarly to proven-in-use tools, the fact that (a particular version of) the tool has been proven correct means that you cannot extend it with domain-specific constructs, limiting the usefulness of the approach for validation with domain experts. The Lustre language used by Scade is essentially a synchronous dataflow language. Consequently, Scade could not sensibly be used for our use case.

Formally Verifying Models This is not a paper on software verification, so we discuss it only briefly. There is a huge community that works on verifying various properties of programs or models. Some of them apply to general-purpose languages and are thus suitable for verifying *implementations* in scenarios similar to the one described in this paper. The canonical example is model checking C with CBMC [36]. The SMACCMPilot project[27] uses the Ivory language [26], a kind of C embedded in Haskell, for implementing high-reliability embedded software. Wasilewski et al. define the requirements of DSLs that use formalisms from automata theory [76]. However, all of these languages are unsuitable for use with our domain experts.

Other approaches work on models expressed with specific modeling languages optimized for expressing systems and properties in a way that makes the verification simpler (an example is PROMELA/Spin [27]). These languages are typically also much too technical for use with non-programmer domain experts. In addition, this category has the same problem of correctly translating to the implementation as the approach described in this paper.

Molotnikov et al. [49] present a partial solution to this problem in the context of the pacemaker challenge [46]. They rely on C extensions to raise the abstraction level of the code to be closer to the pacemaker domain, and then specify (temporal) correctness properties on that level. The actual verification is performed on the level of the C implementation using the aforementioned CBMC. So, similar to the approach in this paper, validation and property specification happens at the model level, but the actual verification is performed on the level of the generated code (the ultimately relevant artifact). The difference to the approach discussed in this paper is is that Molotnikov relies on C and temporal properties; these are not accessible to domain experts.

Formalisation of Language Definitions Whalen and Heimdahl [78] define *requirements* for the use of code generation in safety-critical systems such as formal specification of source and target languages as well as a formally

[27] `https://smaccmpilot.org/`

verified code generator. They provide an example for the former in the form of very simple state-based and imperative languages. However, these are not expressive enough to be used with domain experts. They do not have a provably correct generator. Because of these two limitations this approach does not really satisfy the requirements they propose and is of now help to us. More generally, several works address the correctness of *language* definitions and language implementations. The goal is to ensure soundness of the language definition itself (not just of particular programs/-models, as in the previous paragraph), so once a language designer has finished designing the language, and the LWB reports no more errors, it is guaranteed that *all programs* written with that language are correct. The approaches rely on using more analyzable formalisms for specifying the language implementations themselves. Visser's essay [64] expresses the goals and possible approaches very well. There is work that relies on model checking [45], automated theorem proving [9], and more generally, compiler verification [41]. While these approaches are very useful, it is unrealistic today, in the context of the vast majority of industry projects, to formally verify the definitions of languages that have the size and complexity of those we use with domain experts.

Safety Analysis DSLs DSLs are used for analysing safety properties of systems: DSLs are used to model faults and their propagations [75], to express fault tree analyses [57] or to analyze architectural models for their safety properties [8]. While this approach also uses DSLs in the context of safe system development, they are pure analysis tools; the CSC is not automatically derived from them, so they are a lower risk. By connecting fault tree analyses or failure mode and effects analyses directly with system and test models (M_S/M_T) expressed in the same LWB, benefits regarding productivity and consistency can be achieved compared to using separate, external tools. In fact, Safety.Lab, introduced in [57] is implemented with MPS and could be used for this purpose (as they state in their future work).

9. Discussion

9.1 Justification of Higher Efforts

The additional verification and validation steps summarized in Section 5.5 and Figure 5 lead to additional effort compared to the use of LWBs outside of critical systems. However, the critical systems community accepts that the verification effort is several times higher than in other systems (up to 1,000 USD per LoC [23] compared to 15-50 USD in regular embedded systems [33]). However, the higher effort has to be weighed against the traditional, manual process, taking into account the benefits discussed in Section 3. Anecdotally, the introduction of a LWB and DSLs can reduce the development effort by a factor of 10 or more; and our case study certainly seems to confirm this. 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.

9.2 In-IDE Interpreters as a Central Building Block

Implementing an interpreter directly in the development tool proves to be particularly appealing, because it solves many problems at the same time (reducing overall effort), both in the context of validation and correct implementation.

For validation, it is the basis for a simulator that allows DSL users to interactively explore the models. It can also be used to support the DSL users in testing the (functional) behavior of their models without relying on generation, compilation and deployment, shortening the turnaround time and simplifying the required infrastructure on their computers.

For verification it can act as a redundant execution platform, uncovering errors in the transformations (unrelated but compensating, as well as related errors). Because it does not rely on transformations, it can help reveal errors in the transformation engine itself. Finally, because interpreters used this way usually do not have to be particularly fast or efficient in terms of resource consumption, their implementation is often much simpler than the implementation of generators, which usually do have to generate fast and efficient code. This makes the interpreter implementation less error prone, letting them serve almost as a specification of the semantics; errors are more likely in the generators.

Because of these benefits for validation and correctness, we have been using in-IDE interpreter with many DSLs. Combined with a modular, reusable expression language like KernelF that already comes with an interperter that is tested extensively, makes the approach even more appealing.

An interpreter that is used in the way discussed here of course does not represent the small-step semantics of a lower level execution platform and cannot reproduce timing properties or other non-functional concerns. Consequently, a successful execution on the interpreter does not guarantee correctness of those aspects.

9.3 Qualified Language Workbenches?

A qualified language workbench is one where the language definition facilities and the generation engine can be relied on to be correct.[28] We could rely on its correctness without project-specific mitigations. If we would test the model, language and generators, there would be no need to test the CSC (cf. case B in Figure 2).

Will we have one? We could obtain a qualified language workbench by proving it correct, developing it with a safety-process or by proving it in use.

Proving a practically usable language workbench correct is a major undertaking. It is complex enough to prove prop-

[28] Note that the languages and generators would still be DSL-specific; otherwise we'd use a fixed language tool and thus move to case A of in Figure 2.

erties about a particular language; for an LWB we would either have to prove the correctness of meta languages, or, as part of language development, assemble proofs that a particular language is correct. While progress is being made (for example, integrating proof assistants into the Spoofax language workbench) it is hard to imagine this to be practical any time soon. Another data point that helps us judge the odds of proving an LWBs correct is the availability of compilers that are proven correct. While they do exist, for example for C [41], ML [38] and Ada [59], the verification effort was substantial; and LWB are much more effort, because of their inherent meta programming capability.

Developing a LWB in a safety-process is feasible in principle (even considering that one would have to use the superset of the various domain-specific safety standards mentioned in Section 2.1). However, considering the additional cost incurred such a process, the complexity of a language workbench, the fact that it is still a niche technology and the relatively small overlap between the safety and language communities, we consider this scenario unlikely, too.[29]

Proving a language workbench (not the languages!) in use is the most likely way to end up with a qualified LWB. However, for this to happen, enough relevant projects would have to use one particular version of a particular LWB. Again, considering the small overlap between the safety and language communities, and the fact that the LWBs evolve quickly, this is also unlikely.

What would change? The architecture introduced in this paper treats the the whole stack as a black box. Only one of the risks in Section 5 ("The transformation engine itself has an error.") results from a problem in the LWB. If we were guaranteed that a part of this stack has no errors, how would we test the remaining part of the stack? How would we test the language and its generator? Probably we would still write test cases on model level and run them on the level of the implementation, so not very much would change.

10. Conclusions and Future Work

We proposed an architecture for using LWBs and DSLs in critical software development that mitigates the risks of potentially faulty transformations from DSL-based models to the code-level implementation. We validate the approach with an industrial case study in the healthcare domain and outline the degree to which this approach is feasible in robotics, space, and automotive.

Our future work includes running projects in the space and automotive domains to learn which modifications to the approach are necessary to approval from 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 [45] into MPS.

We are confident that this architecture will help to make LWBs and DSLs admissible for use in real-world projects, thus allowing the critical software industry to reap the benefits of these technologies that have been extensively documented for non-critical domains.

[29] The authors have anecdotally heard about an attempt to develop a code generator in Ada as part of a mission-critical military project; however, a simple template-expanding code generator is a long way from a full-blown language workbench.

Acknowledgments

The authors would like to thank the team at Voluntis and itemis who built the system that underlies the case study. These include Wladimir Safonov, Jürgen Haug, Sergej Koščejev, Alexis Archambault, Nikhil Khandelwal. We would also like to thank Richard Paige and Sebastian Zarnekow for their feedback on drafts of the paper.

References

[1] M. Amrani, B. Combemale, L. Lucio, G. M. K. Selim, J. Dingel, Y. L. Traon, H. Vangheluwe, and J. R. Cordy. Formal verification techniques for model transformations: A tridimensional classification. *Journal of Object Technology*, 14(3):1:1–43, 2015. . URL `http://dx.doi.org/10.5381/jot.2015.14.3.a1`.

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

[3] M. Beine, R. Otterbach, and M. Jungmann. Development of safety-critical software using automatic code generation. Technical report, SAE Technical Paper, 2004.

[4] L. Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

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

[6] M. Broy, S. Kirstan, H. Krcmar, B. Schätz, and J. Zimmermann. What is the benefit of a model-based design of embedded software systems in the car industry? *Software Design and Development: Concepts, Methodologies, Tools, and Applications: Concepts, Methodologies, Tools, and Applications*, page 310, 2013.

[7] T. Bruckhaus, N. Madhavii, I. Janssen, and J. Henshaw. The impact of tools on software productivity. *IEEE Software*, 13(5):29–38, 1996.

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

[9] A. Chlipala. A verified compiler for an impure functional language. In *ACM Sigplan Notices*, volume 45, pages 93–106. ACM, 2010.

[10] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *Acm sigplan notices*, 46(4):53–64, 2011.

[11] 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.

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

[13] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The astrée analyzer. In *Esop*, volume 5, pages 21–30. Springer, 2005.

[14] 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.

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

[16] F.-X. Dormoy. Scade 6: a model based solution for safety critical software development. In *Proceedings of the 4th European Congress on Embedded Real Time Software (ERTS'08)*, pages 1–9, 2008.

[17] S. Erdweg, T. Van Der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, et al. The state of the art in language workbenches. In *International Conference on Software Language Engineering*, pages 197–217. Springer, 2013.

[18] M. Eysholdt. Executable specifications for xtext. Website, 2014. `http://www.xpect-tests.org/`.

[19] S. P. Florence, B. Fetscher, M. Flatt, W. H. Temps, T. Kiguradze, D. P. West, C. Niznik, P. R. Yarnold, R. B. Findler, and S. M. Belknap. Poppl: a patient-oriented prescription programming language. In *ACM SIGPLAN Notices*, volume 51, pages 131–140. ACM, 2015.

[20] 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.

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

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

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

[24] 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.

[25] F. Hermans, M. Pinzger, and A. Van Deursen. Domain-specific languages in practice: A user study on the success factors. In *International Conference on Model Driven Engineering Languages and Systems*, pages 423–437. Springer, 2009.

[26] P. C. Hickey, L. Pike, T. Elliott, J. Bielman, and J. Launchbury. Building embedded systems with embedded dsls. In *ACM SIGPLAN Notices*, volume 49, pages 3–9. ACM, 2014.

[27] G. Holzmann. *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, 2003.

[28] W.-l. Huang and J. Peleska. Exhaustive model-based equivalence class testing. In *IFIP International Conference on Testing Software and Systems*, pages 49–64. Springer, 2013.

[29] E. Jürgens and M. Feilkas. Domain specific languages. 2006.

[30] J. Kärnä, J.-P. Tolvanen, and S. Kelly. Evaluating the use of domain-specific modeling in practice. In *Proceedings of the 9th OOPSLA workshop on Domain-Specific Modeling*, 2009.

[31] L. C. Kats, R. Vermaas, and E. Visser. Integrated language definition testing: enabling test-driven language development. In *ACM SIGPLAN Notices*, volume 46, pages 139–154. ACM, 2011.

[32] R. B. Kieburtz, L. McKinney, J. M. Bell, J. Hook, A. Kotov, J. Lewis, D. P. Oliva, T. Sheard, I. Smith, and L. Walton. A software engineering experiment in software component generation. In *Proceedings of the 18th international conference on Software engineering*, pages 542–552. IEEE Computer Society, 1996.

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

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

[35] 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.

[36] D. Kroening and M. Tautschnig. Cbmc–c bounded model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391. Springer, 2014.

[37] A. Kuhn, G. C. Murphy, and C. A. Thompson. An exploratory study of forces and frictions affecting large-scale model-driven development. In *International Conference on Model Driven Engineering Languages and Systems*, pages 352–367. Springer, 2012.

[38] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. Cakeml: a verified implementation of ml. In *ACM SIGPLAN Notices*, volume 49, pages 179–191. ACM, 2014.

[39] R. Lämmel. Grammar testing. In *Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering*, 2001.

[40] 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*.

[41] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[42] J. Lewis. Cryptol: specification, implementation and verification of high-grade cryptographic applications. In *Proceedings of the 2007 ACM workshop on Formal methods in security engineering*, pages 41–41. ACM, 2007.

[43] G. Liebel, N. Marko, M. Tichy, A. Leitner, and J. Hansson. Assessing the state-of-practice of model-based engineering in the embedded systems domain. In *International Conference on Model Driven Engineering Languages and Systems*, pages 166–182. Springer, 2014.

[44] P. Liggesmeyer and M. Trapp. Trends in embedded software engineering. *IEEE software*, 26(3), 2009.

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

[46] D. Méry, B. Schätz, and A. Wassyng. The pacemaker challenge: Developing certifiable medical devices (dagstuhl seminar 14062). In *Dagstuhl Reports*, volume 4. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014.

[47] 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.

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

[49] Z. Molotnikov, M. Völter, and D. Ratiu. Automated domain-specific c verification with mbeddr. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 539–550. ACM, 2014.

[50] P. Munier. Polyspace®. *Industrial Use of Formal Methods: Formal Verification*, pages 123–153, 2012.

[51] G. J. Myers. *Software Reliability*. John Wiley & Sons, Inc., 1976.

[52] G. J. Myers. A controlled experiment in program testing and code walkthroughs/inspections. *Communications of the ACM*, 21(9):760–768, 1978.

[53] 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.

[54] M. Pajic, Z. Jiang, I. Lee, O. Sokolsky, and R. Mangharam. Safety-critical medical device development using the upp2sf model translation tool. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):127, 2014.

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

[56] D. Ratiu, B. Schaetz, M. Voelter, and B. Kolb. Language engineering as an enabler for incrementally defined formal analyses. In *Proceedings of the First International Workshop on Formal Methods in Soft-*

ware Engineering: Rigorous and Agile Approaches, pages 9–15. IEEE Press, 2012.

[57] 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*, pages 72–82. Springer, 2014.

[58] 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.

[59] V. Santhanam. The anatomy of an faa-qualifiable ada subset compiler. In *ACM SIGAda Ada Letters*, volume 23, pages 40–43. ACM, 2002.

[60] A. Svendsen, G. K. Olsen, J. Endresen, T. Moen, E. Carlson, K.-J. Alme, and Ø. Haugen. The future of train signaling. In *International Conference on Model Driven Engineering Languages and Systems*, pages 128–142. Springer, 2008.

[61] J.-P. Tolvanen, V. Djukić, and A. Popovic. Metamodeling for medical devices: Code generation, model-debugging and run-time synchronization. *Procedia Computer Science*, 63:539–544, 2015.

[62] A. Van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.

[63] 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.

[64] 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.

[65] M. Voelter. Language and ide modularization and composition with mps. In *Generative and transformational techniques in software engineering IV*, pages 383–430. Springer, 2013.

[66] M. Voelter. *Generic tools, specific languages*. TU Delft, Delft University of Technology, 2014.

[67] M. Voelter and S. Lisson. Supporting Diverse Notations in MPS' Projectional Editor. *GEMOC Workshop*.

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

[69] M. Voelter, D. Ratiu, B. Kolb, and B. Schaetz. mbeddr: Instantiating a language workbench in the embedded software domain. *Automated Software Engineering*, 20(3):339–390, 2013.

[70] M. Voelter, D. Ratiu, and F. Tomassetti. Requirements as first-class citizens: Integrating requirements closely with implementation artifacts. In *ACESMB@ MoDELS*, 2013.

[71] M. Voelter, A. v. Deursen, B. Kolb, and S. Eberle. *Using C language extensions for developing embedded software: A case study*, volume 50. ACM, 2015.

[72] M. Voelter, A. van Deursen, B. Kolb, and S. Eberle. Using c language extensions for developing embedded software: A case study. In *OOPSLA 2015*, 2015.

[73] M. Voelter, B. Kolb, T. Szabó, D. Ratiu, and A. van Deursen. Lessons learned from developing mbeddr: a case study in language engineering with mps. *Software & Systems Modeling*, pages 1–46, 2017.

[74] M. Voelter, T. Szabó, and B. Engelmann. *An Overview of Program Analysis using Formal Methods*. Self-published, 2017.

[75] M. Wallace. Modular architectural representation and analysis of fault propagation and transformation. *Electronic Notes in Theoretical Computer Science*, 141(3):53–71, 2005.

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

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

[78] M. W. Whalen and M. P. E. Heimdahl. An approach to automatic code generation for safety-critical systems. In *Automated Software Engineering, 1999. 14th IEEE International Conference on.*, pages 315–318. IEEE, 1999.

[79] J. M. Wing. Computational thinking. *Communications of the ACM*, 49(3):33–35, 2006.

[80] A. Wortmann and M. Beet. Domain specific languages for efficient satellite control software development. In *DASIA 2016*, volume 736, 2016.

[81] H. Wu, J. G. Gray, and M. Mernik. Unit testing for domain-specific languages. In *Domain-Specific Languages, IFIP TC 2 Working Conference, DSL 2009, Oxford, UK, July 15-17, 2009, Proceedings*, pages 125–147, 2009.

A. The PLUTO languages

The exact nature of the DSLs used in PLUTO are not relevant to the contributions of this paper. However, for completeness, we provide an overview over the DSLs here. Note that a discussion of the implementation of the PLUTO languages using MPS is beyond the scope of this paper. We refer the reader to the MPS tutorials[30] or [66].

Main Algorithm The main algorithm controls messages sent to the user and its replies, as well as the timing of those messages and prompts. It also makes high-level decision as to the execution of the algorithm. It is essentially a hierarchical state machine. For complex decisions, it calls into the decision support sublanguage.

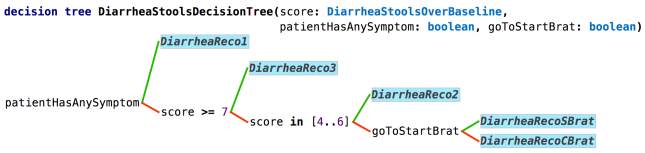

Figure 8. A decision tree; the green/up edges represent `yes`, the red/down edges represent `no`.

```
decision table BpScoreDecisionTable(sys: bpRange, dia: bpRange) =
```

		dia					
		<= 50	[51..90]	[91..95]	[96..100]	[101..109]	>= 110
sys	<= 90	1	1	3	4	5	6
	[91..140]	2	2	3	4	5	6
	[141..150]	3	3	3	4	5	6
	[151..160]	4	4	4	4	5	6
	[161..179]	5	5	5	5	5	6
	>= 180	6	6	6	6	6	6

Figure 9. A decision table that specifically works on ranges of values. Note the compact syntax for range representation.

Decision Support The decision support abstractions can, at a high-level, all be seen as functions: based on a list of arguments, the function returns one or more values. Plain functions are available for arithmetic calculations. However, it is typical for medical decisions to depend on the interactions between several criteria. To make them more readable (and this easier to validate), they are often represented as decision trees (Figure 8) or decision tables. A particular kind of decision table is one that splits two values into ranges and returns a result based on these ranges. Figure 9 shows a table that returns a score based; scores represent standardized severities or risks that are then used in the algorithm. The number types with ranges, and their static checking (see Figure 10), is also an important ingredient to being able to represent the medical domain correctly.

Testing Testing is an important ingredient to the PLUTO languages. For testing functions and function-like abstractions, regular JUnit-style function tests are supported; Figure 12 shows an example. The first of the tests in Figure 12 tests a function with one argument, the second one passes an argument list, and the last one shows how complex data structures, in this case, a patient's replies to a questionnaire,

[30] `https://www.jetbrains.com/mps/concepts/`

```
type temperature: number[36|42]{1}
type measuredTemp: number[35|43]{2}
val T_measured: measuredTemp = 42.22
val T_calibrated: temperature = T_measured * 0.93
```

Error: type number[32.55|39.99]{4} is not a subtype of number[36|42]{1}

Figure 10. Numbers are specified with a range and a precision. The type system checks number ranges and precisions even for simple computations with such values; the figure above shows an error resulting from invalid ranges.

equivalence partition for BPStateMachine

inputs
EventNewBpMeasure(sm: BPStateMachine)

Input	Type	Partitions
inputBaselineDBP	bpRange	<= 90,]90..100], >= 99
inputBaselineSBP	sbpBaseline	<= 150, > 150]
systolic	bpRange	<= 90, > 90
diastolic	bpRange	<= 90, > 90

Figure 11. Equivalence partitions help test complex structures with relevant combinations of values.

```
PASS
function test gradeStools
  given 7 expected 3
  given 6 expected 2
  given 5 expected 2
  given 4 expected 2

PASS
function test DiarrheaStoolsDecisionTree
  given false, 1, true, false   expected   DiarrheaUSRecoLevel1Symptom
  given false, 9, false, false  expected   DiarrheaUSRecoGrade3

PASS
function test checkScreeningQuestion
  given   answers to DiarrheaScreeningQuestionnaire{   expected  true
            dietarySupplements: false
            medication        : true
            hospitalized      : false
          }
```

Figure 12. Function tests call a function (or something function-like, such as a decision tree or table) with the arguments specified after `given`, and then check that the `expected` valued is returned. The `answers` construct represents a user's reply to a questionnaire; it can be seen as an instance of a record.

are passed to the test. The table notations for testing based on equivalence partitions in shown in Figure 11.

Scenario tests (Figure 13) are more involved because they take into account the execution of algorithm over time. They are expressed in the well-known given-when-then style,[31] which is, for example, also supported by the Cucumber test tool.[32] To express the passage of time and occurencess at specific times, the `at` notation is used. The execution of the tests is based on a simulation. The number of steps and the time resolution is derived from the scenario specification.

Simulation The purpose of the simulator is to let HCPs "play" with an algorithm. To this end, the in-IDE interpreter executes algorithms and renders a UI that resembles the one

[31] `https://martinfowler.com/bliki/GivenWhenThen.html`

[32] `https://cucumber.io/`

```
scenario scenario_8
  global timeout:   1 hours
  time granularity: 60 seconds
given
  inputPainBaseline         = 1
  inputPainMedicineDuration = Six
when
  at 0 min: EventInPainMeasure answers to PainMeasureQuestionnaire {
                                 measure: 4
                               }
            EventInPainSymptoms1 answers to PainSymptoms1Questionnaire {
                                   interferingDailyActivities: false
                                   newSite                   : false
                                   interferingAbilityToWalk  : false
                                 }
then
  at 0 hours: assert parent sent message Recommendation(PainRecoSymptom1, Six)
  at 29 min:  assert parent in state     PainMeasure.Ask
  at 30 min:  assert parent in state     PainInitial.Ask
```

Figure 13. Scenarios follow the established given-when-then style:*given* a couple of preconditions, *when* something happens, *then* a set of assertions must hold. Scenarios express the passage of time, as well as points in when something happens or is asserted, using the `at` notation.

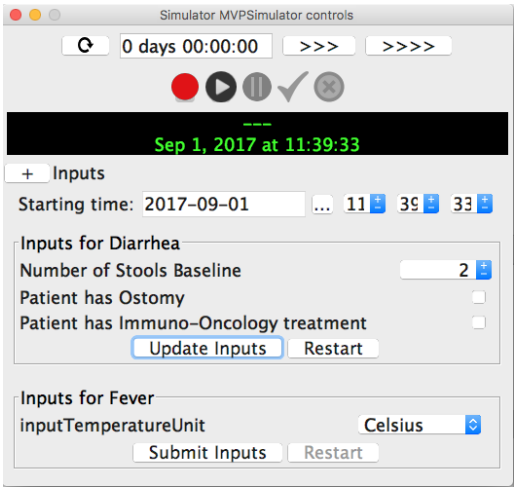

Figure 14. Control panel to configure the simulations.

on the final phone (Figure 15). A set of DSLs is available to style the UI, to some degree, lower-level styling support is available through Javascript and CSS. A control panel lets users configure a particular simulation and also fast-forward in time (Figure 14). There is also a debugger that, while relying on the same interpreter, provides a lower-level view on the execution of algorithms. It is not used by HCPs.

Documentation Generation One important kind of output is the medical protocol, a visualization of the overall algorithm for review by HCPs or associated medical personnel not trained in the use of the PLUTO DSLs. The outputs are too large to show here; they are essentially graphviz-style flow charts with a couple of special notational elements. It is often necessary to highlight specific aspects on the overall algorithm. To this end, the generation of the flow chart can be configured using a DSL (Figure 16). It supports:

- The level of detail (`Deep` in the example)
- The tags that should be included and excluded. Model elements can be tagged, for example, with whether they are part of the default flow or whether they are relevant for complications in the treatment. A generated visualization might want to highlight specific tags.
- Color mappings for tags (e.g., render the case for complications in red)
- Human-readable labels for states or messages in order to make them more understandable for outsides.

The reason why these configurations are represented as models (expressed in their own DSL) as opposed to just configuring a particular visualization through a dialog is that many such configurations exist, and they must be reproduced in bulk, automatically, as the algorithm evolves.

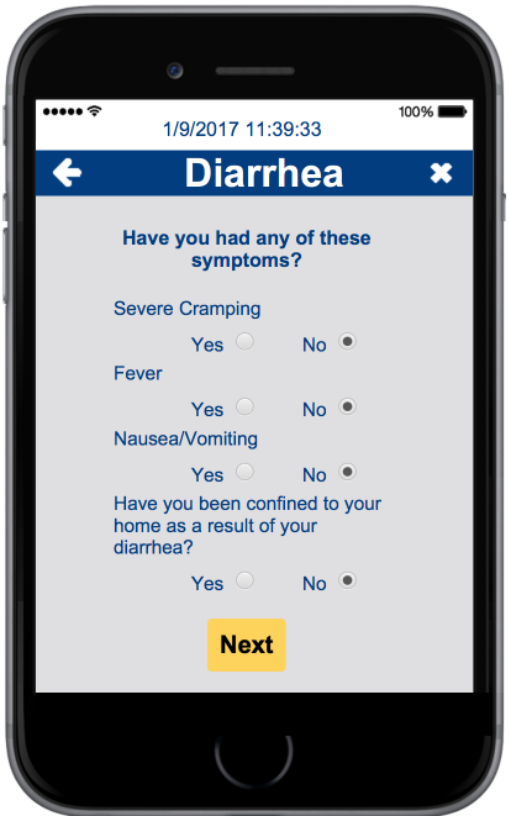

Figure 15. The simulator lets users play with an algorithm. DSLs are available to style the UI. Note that, while an iPhone-style frame is shown, the simulator does *not* run on Apple's iOS simulator.

```
medical protocol for: ComposedStateStoolsOngoingIn24hr
  depth:    Deep
  included: everything
  excluded: #background #log
  colors
  labels:   StoolsOngoingIn24hr TITLE -> 'Diarrhea within 24 h'
            AppRecommendation -> 'Display recommendation "${message}"'
```

Figure 16. Configuration for the generation of medical protocol flow charts.