Asserting Functional Equivalence between C Code and SCADE Models in Code-to-Model Transformations

Jan Toennemann
Adina Aniculăesei
Andreas Rausch
jan.toennemann@tu-clausthal.de
adina.aniculaesei@tu-clausthal.de
andreas.rausch@tu-clausthal.de
Institute for Software and Systems Engineering, TU Clausthal
Clausthal-Zellerfeld, Germany

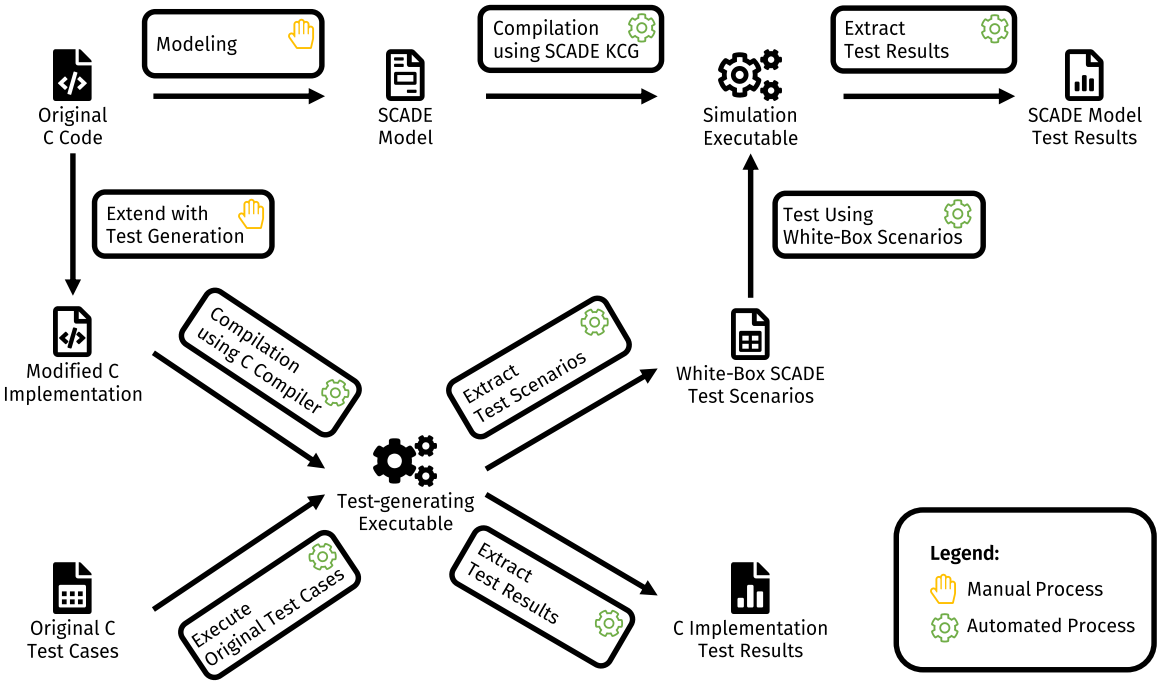

Figure 1: Visualization of the developed concept, asserting functional equivalence between C code and a corresponding SCADE model using the generation of white-box SCADE test scenarios during execution of a slightly modified C implementation

ABSTRACT

Model-based development is on the rise and tool chains employing automated code generation from models using certified code generators are getting increasingly common. We present an approach which enables the reverse operation and creates an ANSYS SCADE model that is functionally equivalent to the C code. The main motivation behind this development is to enable original equipment manufacturers (OEMs) to further use and maintain legacy code in new development environments, rather than having to re-develop the respective functionality from scratch.

While the model transformation itself is performed manually, the testing process is fully automated and enabled the transfer of existing test cases for the C function to the SCADE Test Environment. The presented approach enables white-box testing of the model, requiring the original C implementation and its original test cases as well as a bi-directional mapping of variable names between C code and SCADE model. This is done by extending the original code in a way that generates SCADE test scenarios during runtime, allowing to use these white-box test scenarios to assert functional equivalence of code and model using empirical validation.

SAST '20, October 20–21, 2020, Natal, Brazil

ACM ISBN 978-1-4503-8755-2/20/09…$15.00
https://doi.org/10.1145/3425174.3425213

CCS CONCEPTS

• **Software and its engineering → Model-driven software engineering**; *Abstraction, modeling and modularity*; **Empirical software validation**; • **Computing methodologies → Model verification and validation**; • **Applied computing** → Engineering.

KEYWORDS

equivalence testing, code-to-model transformation, test case generation, C code, ANSYS SCADE model, white-box testing, model-driven engineering, test automation, embedded software

ACM Reference Format:
Jan Toennemann, Adina Aniculăesei, and Andreas Rausch. 2020. Asserting Functional Equivalence between C Code and SCADE Models in Code-to-Model Transformations. In *5th Brazilian Symposium on Systematic and Automated Software Testing (SAST '20), October 20–21, 2020, Natal, Brazil.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3425174.3425213

1 INTRODUCTION

Software is growing increasingly complex and in safety-critical domains, e.g. aerospace and automotive, the developed software is subject to very strict safety requirements.

The complexity of software in safety-critical domains is best observed in projects which have grown organically over the years. In such projects, there is a high chance that legacy code is used in order to keep the overall system running. Often, the legacy code has been developed with toolchains for which technical support might not be provided anymore. However, discarding the legacy code and developing the respective functionality from scratch is in most cases not cost-effective for organisations.

Integration of legacy code with model-based system components and subsystems is a challenge for the OEMs. This is because legacy code has often been implemented directly, only based on informal textual requirements specification, while recent development in safety-critical domains has become more model- and specification-based. In the last years, software engineers in the automotive domain have turned to the model-driven engineering (MDE) approach in order to develop formal models for the requested software system early in the development lifecycle. The software implementation is then tested against these models via back-to-back testing [3], [4].

There are various tools and toolchains which offer additional support with the implementation of the designed formal models via code generation facilities, e.g. ANSYS SCADE [8]. These code generators often come with complex mechanisms which check that the transformation from model to code has been performed correctly. To accomodate the further usage of legacy code and to reap the benefits of the model-based development approach, the legacy code must be transformed into a functionally equivalent formal model in the input language of the chosen MDE toolchain.

Research Question. To address this problem, this paper answers the following research question: *How can C source code be transformed into a functionally equivalent SCADE model?*

Note that to give an answer to the research question, two issues have to be addressed: (1) transformation of C source code into a SCADE model and (2) assessment of the functional equivalence of the two artefacts.

This paper presents an approach to transform legacy C source code into a functionally equivalent SCADE model. We outline a systematic method of transforming C code into a SCADE model and focus on automatically generating appropriate test cases, which we use to assert the functional equivalence of the two artefacts. To evaluate our approach, we have constructed a case study based on an example system, henceforth denoted as system under test (SUT), provided by an industrial project partner. The SUT is part of automotive control software widely deployed to production vehicles and is accompanied by a comprehensive test suite, which provides 100% Modified Condition/Decision Coverage (MC/DC).

One method of assurance for safety-critical systems is validation of the SUT against its requirements. There are various approaches which address the problem of generating test cases from formal requirements specifications, e.g. via model checking [3, 30]. Requirements-based test cases look at the SUT from a black-box point of view, as they specify the test data for the input parameters and the expected value for the output parameter. In this respect, the test cases provided by our industrial partner are black-box test cases.

However, checking whether the SCADE model is functionally equivalent to the C code does not only involve checking that, for the same inputs, the same outputs are produced by the C implementation and by the SCADE model respectively. Instead, it also means checking the paths in the source code and in the model which are exercised by the test inputs. Consider that a set of test inputs activate the same path in the source code during test execution, but different paths in the SCADE model, e.g. due to divergent handling of conditional expressions. In such a case, the SCADE model cannot be considered to be functionally equivalent to the C legacy code. For this purpose, the black-box test cases do not suffice and additional information from a white-box point of view is necessary.

To obtain additional information about the original implementation, we extract the runtime data, e.g. values of local variables in consecutive computation cycles, and use it together with the predefined test input data to derive SCADE test scenarios. On one hand, we use the SCADE test scenarios to exercise and test the SCADE model in the SCADE Test Environment. On the other hand, we use the predefined test cases to test the C implementation of the SUT and to verify the validity of the test suite generated for the SCADE model with regard to the original specification. In order to extract the runtime data, minimal changes have been carried out on the original implementation source code. Neither the test cases nor the test case execution process defined in advance by the test experts of the industrial partner have been altered. Figure 1 shows the overall concept of the developed approach, giving a high-level overview of the generated artefacts and systematic transformations, making a distinction between manual and automated processes.

Paper Outline. This paper is structured as follows: Section 2 gives an overview of related work relevant to the scope of this paper. Section 3 gives an overview of the concept and details the approach, while Section 4 presents the case study and an example appropriate for the scope of this paper. In Section 5 the results are evaluated and discussed. Section 6 concludes this paper with a summary of this work and points out to future directions of research.

2 RELATED WORK

Regarding code-to-model-transformation, Holzmann and Smith described an approach to extract verification models from source code written in ANSI-C [18]. In an earlier work, they already described an automated transfer from ANSI-C to the input language of the Spin model checker [17]. Godefroid et al. used a controlled execution environment called VeriSoft to dynamically analyze the state space and perform verification [16]. Smyth et al. presented an approach for the automatic derivation of Sequentially Constructive Statecharts (SCCharts) from C code. The resulting models could be used to generate code in a different language and for other platforms. Their main reason for the development of this approach was the high maintenance effort required for legacy C code [28]. Izquierdo and Molina described a similar case, defining the Grammar-to-Model Transformation Language (Gra2Mol), a domain-specific language designed to extract models from code. Its purpose is to serve as a text-to-model transformation language suitable to any source code conforming to formal grammar [21].

Rajan et al. described how requirements coverage metrics can be used to determine how well code generated from a model conforms to the actual model by ascertaining behavioral equivalence between the model and the generated code [27]. Gay et al. warned that the use of simple coverage metrics does not ensure a high fault finding rate, with random tests often being more effective than generated ones – though usage of e.g. MC/DC led to "the generation of test suites achieving higher levels of fault detection than random test suites of equal size" [15].

Coverage metrics are an important tool in measuring the validity of a testing process, throughout this document two related coverage metrics will be used. *Modified Condition/Decision Coverage* (MC/DC) requires that for each decision, every condition is shown to affect the outcome independently. This does not presume an exhaustive evaluation of each decision, but it requires at least $n + 1$ test cases for a decision with n conditions. With full MC/DC, the effect of each condition is tested with regard to every other condition [22]. Whalen et al. presented the concept of *observable modified condition/decision coverage* (OMCDC), where MC/DC is combined with a measurement of *observability*. It does not require more test cases for full coverage than traditional MC/DC, but adds a path condition intended to help reveal fault propagation [29]. This is the coverage metric used by the ANSYS SCADE Test Environment.

Model-based development of automotive control software is a core part in the dissertations of Zander-Nowicka [32] and Kugele [23]. While Zander-Nowicka proposes a model-based framework for testing based on MATLAB, Simulink and Stateflow called *MiLEST*, Kugele presents the *COLA automotive approach* of integrated model-based development in combination with a newly created modelling language.

Miller et al. proposed a translator framework to allow for model checking and theorem proving in combination with existing modelling tools and complex systems [24]. Aniculaesei et al. developed a process for automated test case generation and performed a case study on an Automatic Cruise Control (ACC) system, that was later successfully integrated into the SCADE toolchain [5]. The model-checking based approach was successfully embedded into the proprietary ANSYS SCADE development environment and presented a notable step forward in comparison to fully manual test generation [4].

Empirical software validation using various methods has been a topic of research for well over 40 years [19]. Research has shown that the validity of the results obtained through empirical software validation has been improving over the decades [33].With the coverage metrics and the white-box testing approach used in this paper, empirical validation can be performed with a high degree of confidence.

Angius describes the interdependence of formal methods and empirical verification. It is highlighted that ensuring program correctness is a twofold problem, evaluating whether both the logical model as well as the physical implementation conform to the specification. Using formal methods and empirical validation in conjunction with each other bridges domains that are mutually dependent [2]. The works by Boulanger and Dao [9] and Braun et al. [10] provide insight on the topic requirements engineering for large automotive software projects, highlighting which problems occur when working on systems of this scale and giving possible mitigation strategies for lifecycle development.

Hutchinson et al. performed a survey on the state of practice regarding MDE in the industry and found that, along other important factors, integration of the MDE approach into existing process plays an important role in the overall success of the adoption [20]. Whittle et al. found that MDE approaches are already widely used, although often limited to certain components of a larger system [31]. Similar results were observed by da Silva, which found MDE to be on the rise and to be receiving increasing support both with regard to industrial utilizations as well as tool development [13].

3 CONCEPT

The concept we have developed to transform the original C code to a verified implementation in ANSYS SCADE is shown in Figure 1. The majority of the displayed process does not require manual intervention, the two manual steps are described in this section. In this paper we focus on the extension of the C code with automated test case generation and will outline how to transform the code to a model, which still requires several manual design decisions.

3.1 Transferring C Code to SCADE Models

Creating a model from code is a non-trivial task that currently has to be performed manually. During this process, a representation of the original program logic and data flow has to be created using the tools provided by the modeling language. In ANSYS SCADE for example, mathematical operations can be performed using the same operators as in C, while logical branching can either be done using one of the Choice-operators or an If-block. Adapting to the idiomatic concepts of the modeling language might change how some values are computed, but shall not alter the computation result.

Readers unfamiliar with the MDE approach or the SCADE environment are directed to take a look at the SCADE Suite User Manual [14], which introduces both the fundamentals of the development environment and of the modeling language with its intrinsic properties. In general, MDE approaches tend to be seen as

more approachable compared to the manual creation of source code, since the focus is on the actual computation logic rather than on the implementation details in a specific programming language. But transferring a logical model from source code to a model requires the transfer of the underlying concepts from one domain to the other and is bound to several individual design decisions. Just as with writing source code, there are multiple ways to solve the same problem and for any given source code, various different, but still behaviourally equivalent models can be created.

The case study we worked with came from the automotive domain and consisted of a unit extracted from automotive control software currently used in production vehicles. As long as a SCADE model can be produced that is functionally equivalent to the original code, the algorithm presented in the following subsection can be applied to the original C code. Should the code in question not be suitable for iterative execution, it also cannot be modeled in ANSYS SCADE, which requires that the code is free of circular dependencies in the data flow.

When creating both the interface and the local variables in ANSYS SCADE, the same data types as in the original C implementation are to be used. During the model creation, a bi-directional mapping between C variable names and ANSYS SCADE variable descriptors has to be created. To be able to use the resulting scenario files for simulation and testing, the SCADE descriptors used in the mapping must either be the full paths to the variable or an alias file has to be loaded, mapping the variable name to its fully qualified name including the path.

For local variables to be available during testing, they have to be connected to probes and they need to be exported by KCG when generating the testing code. This enables KCG to produce additional context information, forwarding the probes and thus the local variables to the executable output.

When a first iteration of the model has been created, it can already be tested using the generated test suites as covered in the upcoming subsection. With the aid of the white-box test scenarios, development was sped up notably in our case study since remaining modeling errors could be fixed systematically, such that we recommend a combined iterative development approach when transferring the code to a model.

3.2 Extending the C Code with Test Generation

In order to automatically generate the SCADE test scenarios during runtime, several modifications have to be made to the existing C code. While this process can be automated, the modifications required can be performed with low expenditure.

Algorithm 1 provides a pseudocode notation of the modifications made to the existing C implementation used to execute the existing test cases on the C code. The algorithm takes as input a test case denoted as two functions *tc_in* and *tc_out*, mapping a cycle number and a C variable name to an input or output value respectively. Each variable to be set or checked has exactly one defined or expected valuation in any given cycle.

We consider $\texttt{varnames}_{C,in}$ to be the set of input variables and $\texttt{varnames}_{C,out}$ to be the set of output variables for a test case. The set of C variable names that have a valid mapping to SCADE variables is denoted as $\texttt{varnames}_C$. We label the set of C input variables

Input: $tc\_in : \mathbb{N} \times \texttt{varnames}_{C,in} \mapsto \texttt{values}_{C,in}$,
$tc\_out : \mathbb{N} \times \texttt{varnames}_{C,out} \mapsto \texttt{values}_{C,out}$,
$varmap : \texttt{varnames}_C \mapsto \texttt{varnames}_{SCADE}$
Data: $eval\_at : \mathbb{N} \times \texttt{varnames}_C \mapsto \texttt{values}_C$
Result: $tc\_result : \mathbb{N} \times \texttt{varnames}_{C,out} \mapsto [true, false]$
begin
  $i := 0$
  repeat
    foreach $\texttt{var} \in \texttt{varnames}_{C,in}$ **do**
      $\texttt{var} := tc\_in(i, \texttt{var})$
      output "SSM::SET $varmap(\texttt{var})$ $tc\_in(i, \texttt{var})$"
    end
    foreach $\texttt{var} \in \texttt{varnames}_C \setminus \texttt{varnames}_{C,in}$ **do**
      if *end of scope for* var *or end of cycle* **then**
        if $\texttt{var} \in \texttt{varnames}_{C,out}$ **then**
          assert $eval\_at(i, \texttt{text}) = tc\_out(i, \texttt{var})$
        end
        output "SSM::CHECK
         $varmap(\texttt{var})$ $eval\_at(i, \texttt{var})$"
      end
    end
    $i := i + 1$
    output "SSM::CYCLE"
  until *last cycle of input test case*
end

Algorithm 1: Systematic extension of C code to enable the generation of SCADE test scenarios during test case execution.

which are set at the beginning of each cycle during test case execution to be $\texttt{varnames}_{C,in}$ with $\texttt{varnames}_{C,in} \subset \texttt{varnames}_C$ and $\texttt{varnames}_{C,out}$ to be the set of C output variables which are checked at the end of each cycle with $\texttt{varnames}_{C,out} \subset \texttt{varnames}_C$. In addition, we assume $\texttt{varnames}_{C,in} \cap \texttt{varnames}_{C,out} = \emptyset$ such that input variables cannot be output variables at the same time. To not require the variable identifiers in the SCADE model to exactly resemble the ones in the original C code, we introduce the function *varmap*, which maps C variable names to their respective path names in the SCADE model – this mapping can be considered to be a byproduct of the modeling process and usually is available bidirectionally.

Further, we assume a function *eval_at* that maps a cycle number and a C variable name from the set of all C variable names to the evaluation of the corresponding C variable at the end of the given cycle, or at the end of the variable's scope during the execution cycle. The result of the algorithm shall be the same as that of the original test case execution, in that information is returned about whether the expected output variables were produced during the execution given the designated input for the cycle. For this purpose, we have defined a function *tc_result* providing the result of the assertion for a given cycle and output variable name as a boolean value. This value is implicitly set by the assertion and shall always be true, otherwise the original implementation does not pass the used test suite, indicating problems with either the used source code or test cases.

In a cyclic execution pattern, the input variables of the function are set at the very beginning of each cycle to the values specified in the test case. This is directly followed by output in the form "SSM::SET <PATH> <VALUE>", which reflects the same action in a SCADE test scenario. For each of the remaining variables that have a valid mapping to SCADE variables, either output variables or locals, at the end of its scope or at the end of the current cycle, the algorithm prints out a line "SSM::CHECK <PATH> <VALUE>". Through this line it is checked whether the corresponding SCADE variable matches the runtime evaluation of the current C variable at the end of the cycle. Should the C variable be part of the test case output variables, the algorithm asserts that the variable's valuation matches the expected value, as would be done by the original black-box test cases. At the end of each cycle, the proposed algorithm outputs "SSM::CYCLE" to end the cycle. These operations are repeated for each cycle of the test case. The algorithm is applied for every test case in the test suite.

This will generate valid output conforming to a SCADE scenario file for every test case in the original C test suite. Regardless of how the output is created, we recommend saving each test case as a separate `.sss` file for easy integration into the SCADE test environment. Using the SSM::ALIAS <path> <alias> statement, paths can be aliased which can reduce the length of the identifier used in the variable mapping.

The resulting tests also allow for black-box testing by limiting the amount of checked variables to the original test case output variables. Nevertheless, by default, white-box test scenarios are created for the SCADE model, as it is important to not only to test whether the model passes the original test suite, but also to check whether all internal variables are equivalent to their valuation in the C code during the respective computation cycle. Should each variable evaluate to exactly the same value in the respective cycle for a large enough test suite, then it can be said that, with a very high confidence, the code and the model behave functionally equivalent.

4 CASE STUDY

For our case study, we had to migrate an existing legacy C implementation to a new MDE toolchain employing the ANSYS SCADE Suite, a model-based development environment for embedded software with a focus on safe and secure systems. Development in the SCADE Suite can be performed using model-based development, as the graphical diagrams are composed of blocks, connected by edges representing the data flow between the blocks. These diagrams can be converted to their textual Scade 6 [12] representation, which is also a preprocessor step in the code generation process.

The formal background language *Scade 6* is based on Lustre [12], but incorporates features from Esterel [6, 7] and SyncChart [1], functional language arrays and iterators [25] as well as Lucid Synchrone [11]. SCADE features the certified KCG code generator, which generates efficient, deterministic code. The execution order is based on dependencies, not on the diagram's layout or any hidden properties.

The SUT consisted of an automotive control software unit, designed to run inside a feedback loop. Since this is exactly the domain which the ANSYS SCADE Suite is suited for, a manual transfer of the model could be performed after the original implementation was analyzed. Due to the non-disclosure agreement (NDA) signed with the industrial partner, no further details can be given about the unit implementation or the surrounding software architecture. Although we are unable to share details about the original system, the unit on which the case study was performed does surpass four-digit source lines of code (SLOC) and is part of an integrated production vehicle system. In a manual code-to-model transformation process spanning three-digit man hours, small deviations in behavior occurring mainly in edge cases, turned out to be the most time- and thus cost-intensive issues.

As the SUT performed a lot of complex calculations, it was determined that a white-box view into the original system would aid in development and validation. Although a static transfer of the black-box tests for the original system gave away information on whether the function passes the original test cases, it did not allow to deduce exactly where potential modeling errors did happen. Using the white-box scenarios generated during runtime, we were able to effectively identify parts of the model that behave different than the original implementation and adjust the model behavior accordingly.

When transferring from C code to a SCADE model, the local variables declared in the C implementation can be systematically transferred to the model. By performing this mapping, test cases can include variables that are not directly output. These test cases were then generated by modifying the original unit source code, printing a variable's value at the end of its scope or at the end of each cycle. Also including the variables that are set and checked in the original test cases, the program output now provided a white-box view into the system. By adapting the output to the SCADE test scenario definition language, we had effectively created a low-maintenance way to check for the functional equivalence of code and model using the SCADE Test Suite.

In addition to this, we also managed to transfer the original test cases developed for the C code automatically. By embedding the test scenario generation into the unit implementation, the approach is independent from the actual execution of the testing. How the unit's functions are actually triggered is not relevant, only which variables are set to a specific value and which output values are expected for which variable. Since this mapping is inherently present in every defined test case, it does not need to be additionally produced.

Overall, this approach automated a good part of the required workflow and we found it to be generally applicable. It has proven to be adaptable, allowing for the systematic development of ANSYS SCADE models from legacy C code with good test suites.

As we are unable to share the extensive case study performed, we have worked out a small practical example to showcase the approach. Several of the properties exhibited in the original system are present in this example as well, although the scope is severely reduced. Nonetheless, the example in the upcoming subsection serves mainly to highlight which changes are required to the source code, how a code-to-model transformation could look like in this limited scope and which information is provided by the generated test scenarios.

```c
uint8_t voltageViolationDetection(uint32_t d_voltage_component, uint32_t q_voltage_component) {
  /* Output input variable values directly after they have been set */
  WHITEBOX_INPUT_UI32("d_voltage_component", d_voltage_component);
  WHITEBOX_INPUT_UI32("q_voltage_component", q_voltage_component);

  /* Local variables declared only in this scope */
  uint64_t feedback_voltage_vector;
  uint64_t adjusted_voltage_feedback;

  /* Checks to determine current operation point */
  feedback_voltage_vector = ((uint64_t)d_voltage_component * (uint64_t)d_voltage_component)
                          + (uint64_t)q_voltage_component * (uint64_t)q_voltage_component));
  adjusted_voltage_feedback = (feedback_voltage_vector * voltage_adjustment_factor)
                            >> FACTOR_PRECISION;

  /* Check for voltage violation, set current cycle bit and shift register */
  if (adjusted_voltage_feedback > allowed_voltage_feedback) {
    voltage_violation_counter |= 1;
  }
  voltage_violation_counter <<= 1;

  /* Output non-input variable values between last alteration and end of scope */
  WHITEBOX_LOCAL_UI64("d_voltage_component", feedback_voltage_vector);
  WHITEBOX_LOCAL_UI64("q_voltage_component", adjusted_voltage_feedback);

  WHITEBOX_OUTPUT_UI8("voltage_violation_counter", voltage_violation_counter);
  return voltage_violation_counter;
}
```

Listing 1: Example of C source code modified according to Algorithm 1

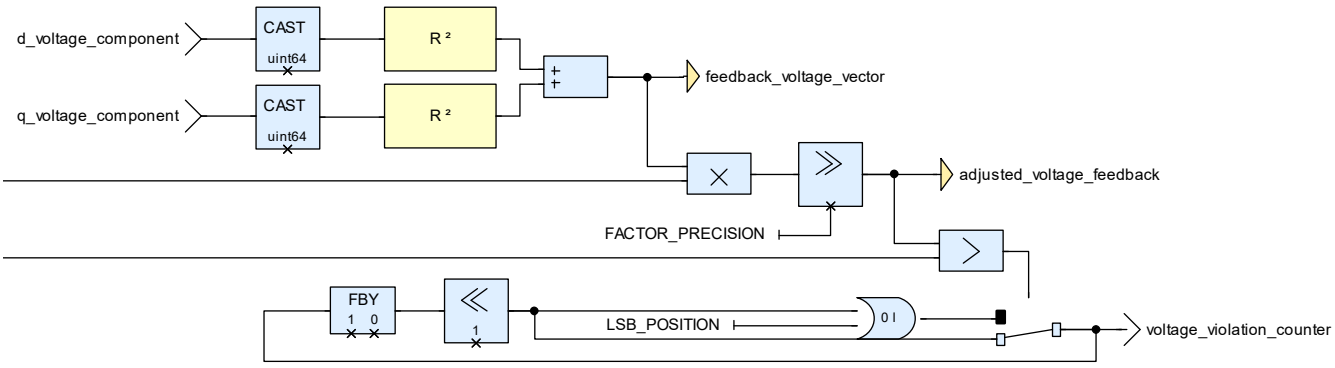

Figure 2: SCADE implementation of the code shown in Listing 1

4.1 A Small Practical Example

Code listing 1 gives a small example of a C function implementation already extended as per Algorithm 1. In the given example, the code snippet is separated as a function, which defines its scope. The function parameters are input variables of the test case, such that they are output directly in the beginning. Two local variables are declared, `feedback_voltage_vector` and `adjusted_voltage_feed`back, for which output statements are added after their last alteration, but before the end of the scope. The function returns one variable, `voltage_violation_counter`, which is not part of the local scope but is used as output for the test case and as such is returned after alteration and before the end of the cycle. Although explicitly construed to showcase the approach, this example resembles a rudimentary supervision function for the detection of voltage violations in a system employing direct-quadrature-zero

Status	Step	Name	Actual Value	Expected Value
✓	1	C_Transfer/voltage_violation_counter	0	0
✓	1	C_Transfer/feedback_voltage_vector_probe	384605000000	384605000000
✓	1	C_Transfer/adjusted_voltage_feedback_probe	2115327500000	2115327500000
✓	2	C_Transfer/voltage_violation_counter	1	1
✓	2	C_Transfer/feedback_voltage_vector_probe	414050000000	414050000000
✓	2	C_Transfer/adjusted_voltage_feedback_probe	2277275000000	2277275000000
✓	3	C_Transfer/voltage_violation_counter	2	2
✓	3	C_Transfer/feedback_voltage_vector_probe	384605000000	384605000000
✓	3	C_Transfer/adjusted_voltage_feedback_probe	2115327500000	2115327500000
✓	4	C_Transfer/voltage_violation_counter	5	5
✓	4	C_Transfer/feedback_voltage_vector_probe	414050000000	414050000000
✓	4	C_Transfer/adjusted_voltage_feedback_probe	2277275000000	2277275000000

Figure 3: Excerpt of test results when executing generated white-box tests on the SCADE model in Figure 2

transformation on three-phase currents, utilized for motor control [26].

The original source code for this scope block has only been altered by application of the algorithm in the addition of lines 2–3, 22–23 and 25 calling header macros prefixed with `WHITEBOX`. The macros expect a string representing the variable name and the corresponding value, each macro is dependent on the data type and as such has a postfix determining the type. We used header macros that write to the standard output device (stdout), since the original function did not produce any output there and it could easily be captured and rewritten to a file.

Note that formally, this example defines: $\texttt{varnames}_{C,in}$ = {`d_component`, `q_component`}, $\texttt{varnames}_{C,out}$ = {`voltage_violation_counter`} and $\texttt{varnames}_{C}$ = $\texttt{varnames}_{C,in} \cup \texttt{varnames}_{C,out} \cup$ {`feedback_voltage_vector`, `adjusted_voltage_feedback`}.

For variables that are considered to be input, output shall occur directly after the variable is altered. Assertion of correct output values are part of the original test execution process already and require no modifications. The information on whether the test case is completely passed by the original implementation is still essential to the process. Variables that are to be checked and that not part of the local execution scope are to be output at the end of each cycle. In our implementation using the header definitions, the macros `WHITEBOX_LOCAL_<type>` and `WHITEBOX_OUTPUT_<type>` produce `SSM::CHECK` statements while the `WHITEBOX_INPUT` macro takes the same parameters, but outputs `SSM::SET` in the beginning. The macro for local variables is simply an alias for the output macro, added for better readability.

Figure 2 shows an excerpt of a SCADE model that shall be tested for functional equivalency. This model is a result of the manual code-to-model transfer process. Most of the used building blocks are basic arithmetic operators or comparisons that resemble their counterpart in the source code, since SCADE uses native C types, even the numeric cast operations are the same. The main difference of code and model is the use of the FBY-operator in SCADE, which allows to reuse the variable value from a previous iteration. In this example, it is used to make the `voltage_violation_counter` stateful, in a way similar to C global variables preserving state over several function calls and iterations. Note that the variables `voltage_adjustment_factor` and `allowed_voltage_feedback` are defined outside of the function scope in the C code and thus the model references these from outside the shown scope as well.

By executing a test case using the modified implementation from Listing 1, valid SCADE test files are generated which can be executed in the SCADE test environment. In Figure 3 we show an excerpt of four test cycles in the SCADE test environment using the test scenario shown in Listing 2 as generated by the modified implementation.

In the given figure, the output of the function is as expected and the values of the local variables conform to those in the C implementation as well. The original black-box test case used here tests the behavior around the boundaries of the maximum allowed feedback voltage, checking for the correct detection of errors, register shifting and application of bitwise operations. The desired behavior is thus, in this case, accurately reflected in the manually created SCADE model.

Using the detailed method, the test case could be transferred automatically and extended with runtime information, namely the actual values of the local variables. As long as the variable mapping as well as the set of input and output variables does not change, no further modifications to the original source code are required. Any number of tests can be run and valid SCADE test scenarios will be output, allowing for the automated transfer of a large test suite easily.

5 DISCUSSION

Before starting the project, we did extensive research on the topic of code-to-model-transformation, program transformation in general as methods to assert functional equivalency of code. We knew that this was a rather unusual domain and that model-to-code-transformations were a lot more common, as every major model-based development suite ships with a certified code generator. But that code-to-model transformations were not only uncommon, but research about the topic was very rare and scarce limited us in the beginning. We found the developed approach to be a resourceful way to solve the equivalence testing problem and having the white-box information available helped immensely during the development of the function.

In our experiments we have seen that the OMCDC coverage of the SCADE tests correlates with the MC/DC coverage of the

```
SSM::set C_Transfer/d_voltage_component 443000
SSM::set C_Transfer/q_voltage_component 434000
SSM::check C_Transfer/feedback_voltage_vector_probe 384605000000
SSM::check C_Transfer/adjusted_voltage_feedback_probe 2115327500000
SSM::check C_Transfer/voltage_violation_counter 0
SSM::cycle
SSM::set C_Transfer/d_voltage_component 455000
SSM::set C_Transfer/q_voltage_component 455000
SSM::check C_Transfer/feedback_voltage_vector_probe 414050000000
SSM::check C_Transfer/adjusted_voltage_feedback_probe 2277275000000
SSM::check C_Transfer/voltage_violation_counter 1
SSM::cycle
SSM::set C_Transfer/d_voltage_component 443000
SSM::set C_Transfer/q_voltage_component 434000
SSM::check C_Transfer/feedback_voltage_vector_probe 384605000000
SSM::check C_Transfer/adjusted_voltage_feedback_probe 2115327500000
SSM::check C_Transfer/voltage_violation_counter 2
SSM::cycle
SSM::set C_Transfer/d_voltage_component 455000
SSM::set C_Transfer/q_voltage_component 455000
SSM::check C_Transfer/feedback_voltage_vector_probe 414050000000
SSM::check C_Transfer/adjusted_voltage_feedback_probe 2277275000000
SSM::check C_Transfer/voltage_violation_counter 5
SSM::cycle
```

Listing 2: Excerpt of a test scenario generated by the modified implementation

C code test suite. With 100% MC/DC in the original test suite we could achieve 100% OMCDC in the generated SCADE tests as well. Following the original case study, the developed approach could be generalized, such that it could be applied to future projects easily.

The initial intent was to automatically transfer the existing test cases to SCADE scenario files, independent of the way the testing is performed. Using runtime information allowed us to hook into the processes that were being executed directly, regardless of how they were triggered. This also enabled us to extract even more valuable information showing not only input-output equivalence, but providing a full assertion of functional equivalence.

Should black-box tests be preferred, the test scope can be reduced to the original test suite. With the mapping reduced to the input and output variables of the original test case only, the algorithm can be further simplified and the modifications required to the original source are reduced even further.

6 CONCLUSION AND FUTURE WORK

We acknowledge that the domain of code-to-model-transformation is still rather small, but are convinced that it is a viable solution with regard to the legacy problem. When the toolchain undergoes a major change, in our example switching from a code-oriented development approach to a model-based one, supporting an older toolchain just for legacy modules becomes increasingly inefficient.

Transferring an implementation from one language to another can be automated, source-to-source transformations between high-level languages are performed by transpilers. These tend to focus on a mapping of the abstract syntax trees (ASTs) between to languages, adapting the syntax of one language to the other. For languages that differ in the programming paradigm, some transfer of concepts might be introduced.

The formal language behind the ANSYS SCADE Suite is a synchronous, declarative programming language. By manually creating a model based on the C code, we have effectively manually transferred a program from the C programming language into the Scade 6 formal language. We have then shown that it is possible to automate the testing for functional equivalency using empirical validation that can be used fully independent of the way the test cases are written or executed.

The listed Algorithm 1 has until now only been performed manually. Modifications required to the original source code are minimal and straightforward and can usually be carried out in a matter of minutes. But still, manual processes tend to be error-prone. The algorithm has yet to be implemented, allowing for automatic adaptation of the source code. This could for example be done using a combination of static code analysis or by running the executable in an environment providing additional debugging hooks. This could also enable to automatically determine whether the code can be converted to a SCADE model or not, e.g. because of incompatible control flow.

Ultimately, an automated transfer of C code to ANSYS SCADE models could be developed. For this, the subset of C implemented by the Scade 6 formal language needs to be identified exactly and a full mapping of the C subset and the Scade 6 language needs to be performed. By limiting the subset further, parts of this transformation could be developed independently, allowing for the automated modelling of different concepts, which ideally could be merged together.

REFERENCES

[1] Ch. André. 1995. SyncCharts: A visual representation of reactive behaviors.

[2] Nicola Angius. 2019. On the Mutual Dependence Between Formal Methods and Empirical Testing in Program Verification. *Philosophy & Technology* 33, 2 (jul 2019), 349–355. https://doi.org/10.1007/s13347-019-00364-9

[3] Adina Aniculaesei, Falk Howar, Peer Denecke, and Andreas Rausch. 2018. Automated Generation of Requirements-Based Test Case for an Adaptive Cruise Control System. In *2018 IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST)*. 11–15. https://doi.org/10.1109/VST.2018.8327150

[4] Adina Aniculaesei, Andreas Rausch, and Andreas Vorwald. 2019. Automated Generation of Requirements-Based Test Cases for an Automotive Function using the SCADE Toolchain. In *11th International Conference on Adaptive and Self-Adaptive Systems and Applications (ADAPTIVE 2019)*.

[5] Adina Aniculaesei, Andreas Rausch, and Andreas Vorwald. 2019. Using the SCADE Toolchain to Generate Requirements-Based Test Cases for an Adapative Cruise Control System. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. 502–512. https://doi.org/10.1109/MODELS-C.2019.00079

[6] Gérard Berry. 2000. The Foundations of Esterel. In *Proof, Language and Interaction: Essays in Honour of Robin Milner* (2000-05-01), C. Stirling G. Plotkin and M. Tofte (Eds.). 425–454.

[7] Gérard Berry. 2004. The Esterel v5 Language Primer Version v5.91. (01 2004).

[8] Gérard Berry. 2007. SCADE: Synchronous Design and Validation of Embedded Control Software. In *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, S. Ramesh and Prahladavaradan Sampath (Eds.). Springer Netherlands, Dordrecht, 19–33.

[9] Jean-Louis Boulanger and Van Quang Dao. 2008. Requirements engineering in a model-based methodology for embedded automotive software. In *2008 IEEE International Conference on Research, Innovation and Vision for the Future in Computing and Communication Technologies*. IEEE. https://doi.org/10.1109/rivf.2008.4586365

[10] Peter Braun, Manfred Broy, Frank Houdek, Matthias Kirchmayr, Mark Müller, Birgit Penzenstadler, Klaus Pohl, and Thorsten Weyer. 2010. Guiding requirements engineering for software-intensive embedded systems in the automotive industry. *Computer Science - Research and Development* 29, 1 (oct 2010), 21–43. https://doi.org/10.1007/s00450-010-0136-y

[11] Paul Caspi, Grgoire Hamon, and Marc Pouzet. 2008. Synchronous Functional Programming with Lucid Synchrone. In *Modeling and Verification of Real-Time Systems*. ISTE, 207–247. https://doi.org/10.1002/9780470611012.ch7

[12] Jean-Louis Colaco, Bruno Pagano, and Marc Pouzet. 2017. SCADE 6: A formal language for embedded critical software development (invited paper). In *2017 International Symposium on Theoretical Aspects of Software Engineering (TASE)*. IEEE. https://doi.org/10.1109/tase.2017.8285623

[13] Alberto Rodrigues da Silva. 2015. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures* 43 (oct 2015), 139–155. https://doi.org/10.1016/j.cl.2015.06.001

[14] Esterel Technologies S.A.S. [n.d.]. *SCADE Suite User Manual*. ANSYS Inc.

[15] Gregory Gay, Matt Staats, Michael Whalen, and Mats P. E. Heimdahl. 2015. The Risks of Coverage-Directed Test Case Generation. *IEEE Transactions on Software Engineering* 41, 8 (aug 2015), 803–819. https://doi.org/10.1109/tse.2015.2421011

[16] Patrice Godefroid, Robert S. Hanmer, and Lalita Jategaonkar Jagadeesan. 2002. Systematic software testing using VeriSoft-An analysis of the 4ESS™ heart-beat monitor. *Bell Labs Technical Journal* 3, 2 (aug 2002), 32–46. https://doi.org/10.1002/bltj.2103

[17] Gerard J. Holzmann. 2000. Logic Verification of ANSI-C Code with SPIN. In *SPIN Model Checking and Software Verification*. Springer Berlin Heidelberg, 131–147. https://doi.org/10.1007/10722468_8

[18] Gerard J. Holzmann and Margaret H. Smith. 2001. Software model checking: extracting verification models from source code. *Software Testing, Verification and Reliability* 11, 2 (2001), 65–79. https://doi.org/10.1002/stvr.228

[19] William E. Howden. 1979. Empirical studies of software validation. *Microelectronics Reliability* 19, 1-2 (jan 1979), 39–47. https://doi.org/10.1016/0026-2714(79)90360-3

[20] John Hutchinson, Mark Rouncefield, and Jon Whittle. 2011. Model-driven engineering practices in industry. In *Proceeding of the 33rd international conference on Software engineering - ICSE '11*. ACM Press. https://doi.org/10.1145/1985793.1985882

[21] Javier Luis Cánovas Izquierdo and Jesús García Molina. 2012. Extracting models from source code in software modernization. *Software & Systems Modeling* 13, 2 (sep 2012), 713–734. https://doi.org/10.1007/s10270-012-0270-z

[22] Hayhurst Kelly J., Veerhusen Dan S., Chilenski John J., and Rierson Leanna K. 2001. *A Practical Tutorial on Modified Condition/Decision Coverage*. Technical Report.

[23] Stefan Kugele. 2012. *Model-Based Development of Software-intensive Automotive Systems*. Dissertation. Technische Universität München, München. https://mediatum.ub.tum.de/node?id=1110104

[24] Steven P. Miller, Michael W. Whalen, and Darren D. Cofer. 2010. Software model checking takes off. *Commun. ACM* 53, 2 (feb 2010), 58–64. https://doi.org/10.1145/1646353.1646372

[25] Lionel Morel. 2007. Array Iterators in Lustre: From a Language Extension to Its Exploitation in Validation. *EURASIP Journal on Embedded Systems* 2007 (2007), 1–16. https://doi.org/10.1155/2007/59130

[26] Colm J. O'Rourke, Mohammad M. Qasim, Matthew R. Overlin, and James L. Kirtley. 2019. A Geometric Interpretation of Reference Frames and Transformations: dq0, Clarke, and Park. *IEEE Transactions on Energy Conversion* 34, 4 (dec 2019), 2070–2083. https://doi.org/10.1109/tec.2019.2941175

[27] Ajitha Rajan, Michael Whalen, Matt Staats, and Mats P. E. Heimdahl. 2008. Requirements Coverage as an Adequacy Measure for Conformance Testing. In *Formal Methods and Software Engineering*. Springer Berlin Heidelberg, 86–104. https://doi.org/10.1007/978-3-540-88194-0_8

[28] Steven Smyth, Stephan Lenga, and Reinhard von Hanxelden. 2016. Model Extraction of Legacy C Code in SCCharts. In *7th International Symposiumon Leveraging Applications of Formal Methods, Verification and Validation*.

[29] M. Whalen, G. Gay, D. You, M. P. E. Heimdahl, and M. Staats. 2013. Observable modified condition/decision coverage. In *2013 35th International Conference on Software Engineering (ICSE)*. 102–111.

[30] M.W Whalen, A. Rajan, and M.P.E. Heimdahl. 2006. Coverage Metrics for Requirements-Based Testing. In *Proceedings of International Symposium on Software Testing and Analysis*. ACM, 25–36. https://doi.org/10.1145/1146238.1146242

[31] Jon Whittle, John Hutchinson, and Mark Rouncefield. 2014. The State of Practice in Model-Driven Engineering. *IEEE Software* 31, 3 (may 2014), 79–85. https://doi.org/10.1109/ms.2013.65

[32] Justyna Zander-Nowicka. 2009. *Model-based testing of real-time embedded systems in the automotive domain*. Ph.D. Dissertation. https://doi.org/10.14279/depositonce-2126

[33] Marvin V. Zelkowitz. 2007. Techniques for Empirical Validation. In *Empirical Software Engineering Issues. Critical Assessment and Future Directions*. Springer Berlin Heidelberg, 4–9. https://doi.org/10.1007/978-3-540-71301-2_2