Achieving ISO 26262 Compliance with Reactis

Facilitate walk-throughs and inspections. Track coverage: Statement, Branch, MC/DC. Check software safety requirements via semi-formal verification. Back-to-back testing: test conformance of code to model.

RSITR 4.3
October 10, 2015

www.reactive-systems.com
## Contents

1 Introduction 1

2 Concepts Underlying ISO 26262 1
  2.1 Automotive Safety Integrity Level 2
  2.2 V Model 3
  2.3 Model-Based Design 4

3 Reactis 5
  3.1 Simulator 5
  3.2 Reactis Tester 6
  3.3 Reactis Validator 6

4 ISO 26262 Tasks Supported by Reactis 7
  4.1 ISO 26262-6: Product Development at the Software Level 7
  4.2 Architectural Design 9
  4.3 Unit Design and Implementation 10
  4.4 Testing Methods and Metrics 11
  4.5 Unit Testing 11
    4.5.1 Unit Testing Methods 12
    4.5.2 Unit Test Generation 14
    4.5.3 Unit Test Coverage Metrics 14
  4.6 Integration Testing 15
  4.7 Verification of Software Safety Requirements 16

5 An ISO 26262 Process using Reactis 17

6 Requirements-Based Testing with Reactis 18

7 Back-to-Back Testing with Reactis 20
  7.1 Back-to-Back Testing with Reactis for C 21
  7.2 Back-to-Back Testing with the Reactis for C Plugin 23
  7.3 Back-to-Back Testing with Reactis Runtests 24
  7.4 Back-to-Back Testing with External Tools 25
  7.5 Errors During Testing 27
  7.6 Comparing Outputs 27
  7.7 Evaluating Coverage Results 27

8 Conclusions 28
About Reactive Systems, Inc.

Reactive Systems, founded in 1999, is a privately held company based in Cary, NC. The company’s Reactis product line provides automated testing and validation tools to support the development of embedded control software. Reactis, Reactis for C Plugin, Reactis for EML Plugin, Reactis Model Inspector, and Reactis for C support model-based design with Simulink, Stateflow, Embedded MATLAB, and C code. Reactis Tester automatically generates comprehensive yet compact test suites from a Simulink model or C code. Reactis is used at companies worldwide in the automotive, aerospace, and heavy-equipment industries.

Reactive Systems, Inc.
341 Kilmayne Dr.
Suite 101
Cary, NC 27511
USA

Tel.: +1 919-324-3507
Fax: +1 919-324-3508

Web: www.reactive-systems.com
E-mail: info@reactive-systems.com
Abstract

This white paper discusses how Reactis®, an automated testing and validation tool, may be used to comply with the ISO 26262 standard. The standard prescribes a system of steps to manage the functional safety of automotive electronics. Part 6 (ISO 26262-6) addresses product development at the software level and is the focus of this paper. When using a model-based design process employing MATLAB®/Simulink®/Stateflow®, Reactis automates a number of the verification activities mandated by ISO 26262. Reactis offers a number of model navigation capabilities that facilitate design walk-throughs and inspections at both the architectural and unit levels. The Reactis Validator component lets you formalize safety requirements as assertions and then check for violations using semi-formal verification. These checks can be performed on both architectural design models and unit design models. Reactis Tester can automatically generate test suites that aim to maximize statement, branch, and modified condition/decision (MC/DC) coverage. Finally, Reactis offers extensive support for back-to-back testing (in which the behavior of code is compared to the behavior of a model).

1. Introduction

Safety-critical functions in automobiles are increasingly being performed by programmable electronic systems. Systems which were formerly controlled using other means (mechanical, hydraulic, pneumatic, electrical, etc.) are ever more likely to be controlled by embedded software. In addition, automakers continue to add new safety features, such as electronic stability control and autonomous emergency braking. These trends are driving an explosive growth in both the amount and complexity of safety-critical software embedded in the typical car. This in turn is making it ever more difficult to minimize the risk posed by malfunctioning embedded automotive software.

ISO 26262 is an international standard (published in 2011) which addresses the safety of electrical and electronic systems within road vehicles. Part 6 of the standard (a.k.a. ISO 26262-6) prescribes best practices for ensuring the safety of the software component of a safety-related system. In the rest of this document, we will show how Reactis can help you comply with the ISO 26262-6 software validation requirements.

2. Concepts Underlying ISO 26262

ISO 26262 prescribes a system of steps to manage the functional safety of automotive electrical/electronic (E/E) systems. It is a road vehicle-specific adaptation of IEC 61508, and targets the safety of both the hardware and software components of automotive E/E systems.
ISO 26262 is based on the concept of a safety lifecycle, shown in Figure 1, which consists of 6 phases: management, development, production, operation, service, and decommission. The goal of the standard is to maximize product safety by requiring specific steps to be taken during each of the phases. This ensures that safety is taken into consideration from the earliest conception of a vehicle to the point when the vehicle is retired from use. This document focuses primarily on the development phase, since this is the step in which embedded software is designed, developed, and validated.

2.1. Automotive Safety Integrity Level

A second key concept in ISO 26262 is the automotive safety integrity level (ASIL), a measure of the risk imposed by a specific system component. As risk increases, more stringent methods must be employed to ensure safety. As shown in Figure 2, there are four ASIL values, named A-D, in which A is the minimum amount of risk, and D is the maximum. The ASIL for each component in a system is determined by three factors: severity, probability, and controllability.

Severity is a measure of the health consequences of an event. There are four classes of severity, ranging from no injuries to life-threatening injuries.

Probability is the likelihood of the conditions under which a particular failure would result in a safety hazard. The probability of each condition is ranked on 5 point scale ranging
from incredible to highly probable. For example, a failure of the headlights would result in a hazard when driving at night, when raining, or during other conditions which result in poor visibility — which would be considered highly probable due to the regular occurrence of these conditions.

**Controllability** is a measure of the probability that harm can be avoided when a hazardous condition occurs, either due to actions by the driver, or by external measures. If the brakes fail to engage when the brake pedal is pressed, for example, the driver could use the emergency brake instead. The controllability of a hazardous situation is ranked on a four point scale from controllable in general to difficult to control or uncontrollable.

Once the severity, probability, and controllability have been determined, Table 4 of Part 3 (ISO 26262-2) is used to determine the ASIL.

### 2.2. V Model

![V Model Diagram](image)

Figure 3: The V-model used in ISO 26262 software development.

The software development phase in ISO 26262 is subdivided into sub-phases according to a V-Model, as shown in Figure 3. The “V” shape is due to the fact that the testing and verification steps are performed in reverse order from design and implementation. Reactis can be used during each of the testing and verification steps.
2.3. Model-Based Design

The usefulness of mathematical models has been long-recognized in domains such as aerospace, civil, and electrical engineering. Model-based software design attempts to apply this concept to the software engineering domain. While ISO 26262 does not require the use of model-based development, the value and importance of the model-based engineering paradigm is emphasized in Annex B of ISO 26262-6.

Figure 4: Model-based development using Simulink/Stateflow and C.

Figure 4 shows an idealized view of a model-based development process using Simulink/Stateflow and C, which consists of three phases.

In the first phase, requirements are gathered. Many of these requirements will be logical in nature, such as mutually exclusive conditions, limits on values which are sent to an output device, or actions which must be performed in a specific order.

In the second phase, an executable model is developed based on the requirements. This model is tested as much as possible to ensure that it satisfies all the requirements before proceeding to the next phase.

In the third phase, C code is produced from the model. This code may consist of components which were written by hand or automatically generated. The behavior of this code must be tested to ensure that its behavior matches the behavior of the model.

The model-based development process offers a number of advantages over traditional approaches.

First, since the models are executable, testing can be done during the design phase by testing the model against its requirements specification, by using a tool such as Reactis Validator. This allows design flaws to be caught and fixed earlier in the development process. In a process which uses non-executable designs, it’s likely that such flaws will not be discovered until after the implementation phase, making the cost of fixing them significantly higher.

Second, since the models are graphical, they serve as visual representations of system structure and data-flow. A control system expressed as a Simulink model, for example, is typically much easier to comprehend than the corresponding design expressed in a textual language.

Third, executable models make it possible to automate the implementation testing to a large degree. Tools such as Reactis Tester can be used to automatically compare the implementation behavior against the model. When non-executable designs are used, on the other
hand, the process of generating tests based on the design is more complicated and involves much more manual labor.

Fourth, when design issues are found, the executable models can be easily tweaked and then re-tested. The executable nature of the model also makes it an excellent platform for experimenting with different design alternatives. Without the ability to execute designs, design alternatives can only be explored through a more cumbersome and error-prone process of thought experimentation.

3. Reactis

Reactis is an automated debugging and test-generation tool which discovers defects in Simulink/Stateflow models and C programs using a simulate/test/validate paradigm. Using Reactis, engineers may:

- Generate test suites from Stateflow/Simulink models or C code. The tests comprehensively exercise all parts of a model or program (structural testing).
- Find run-time errors, such as memory errors, overflow errors, divide-by-zero errors, etc.
- Interactively execute a model or program while tracking progress toward full coverage of a variety of coverage targets including MC/DC targets.
- Perform functional tests to determine if a model or program can ever violate any of its requirements, including safety requirements.
- Replay a specific execution sequence which triggers a defect in order to understand, diagnose, and fix a bug.
- Perform back-to-back comparisons of model and code to ensure that they are functionally equivalent.

Reactis consists of three basic components, as shown in Figure 5. These components are Simulator, Tester, and Validator.

3.1. Simulator

*Reactis Simulator* is an interactive execution environment which supports debugging and visualization. In addition to basic debugging features, such as single-stepping and breakpoints, Simulator provides a number of advanced debugging features, including reverse-stepping and the ability to plot the values of a variable or signal over time. Simulator also performs a large number of runtime error checks as it executes a model/program, so that bugs such as reading from memory which has been de-allocated trigger an immediate pause in execution. When an error occurs, you can step backward while inspecting variable values, making it easier to trace an error back to its original source.

In addition to debugging, Simulator also allows you to visualize test coverage and inspect models or code. Colored annotations indicate the coverage status of all coverage targets in the model/code, including MC/DC targets. This makes it easy to comprehend which parts of a program require additional testing.
3.2. Reactis Tester

Reactis Tester uses a patented technique known as guided simulation to generate test suites which exhaustively exercise a model or program. The test suites produced by Reactis are both comprehensive and compact. They are comprehensive in the sense that they attempt to cover as much of the program structure as possible, as measured by structural testing metrics including statement coverage, decision coverage, condition coverage, and modified condition/decision coverage (MC/DC). The test suites are compact because test steps which do not lead to covering a previously-uncovered target (such as a statement which was not executed during any of the previous tests) are pruned from test suite as it is generated.

3.3. Reactis Validator

Reactis Validator allows you to perform functional testing of program requirements, including safety properties. When viewing a model in Reactis, you can annotate the model with assertions and user-defined targets. These annotations are stored in a separate file so that the underlying model is not modified. An assertion is a universal property which should always be true, such as requiring that the brake and cruise control of a vehicle never be engaged at the same time. A user-defined target is an existential property which should become true at least once during at least one test. For example, a target would be used to ensure that the vehicle speed reaches 75 miles per hour at least once during the test of a cruise control unit.

1United States Patent 7,644,398 titled *System and Method for Automatic Test-Case Generation for Software*
4. ISO 26262 Tasks Supported by Reactis

The ISO 26262 standard consists of ten parts, enumerated as ISO 26262-1 to ISO 26262-10. These parts cover the phases shown in Figure 1. The part of the standard to which Reactis can best be applied is Part 6, Product development at the software level. This part specifies how vehicular software components should be developed in order to minimize safety risks.

4.1. ISO 26262-6: Product Development at the Software Level

ISO 26262-6 specifies what steps should be taken during the development of software components to ensure product safety. The requirements given in ISO 26262-6 fall into seven phases, which are listed in Figure 7 and described below.

Clause 5. Initiation of software development. During this phase, the software development process to be used must be selected, as well as which tools will be used. If you are using model-based development in your organization, then you will be required to create a devel-
Clause 5. Initiation of software development.

Clause 6. Safety requirements specification. The primary objective of the safety requirements specification phase is to identify all software-based functions which could potentially impact safety by either (a) directly producing an unsafe state, such as a software module which controls traction while braking, or (b) failing to correctly handle a hardware or software fault. The requirements for each identified software component must be determined, including such things as timing requirements and the interface between the software component and other system components.

Clause 7. Architectural design. During this phase, a high-level design for each software component is developed. The conformance of the architectural design to the safety requirements must be verified. If the architectural design is in the form of a Stateflow/Simulink model, then Reactis can do this for you automatically, once Reactis Validator objectives have been created for each safety requirement.

Clause 8. Software unit design and implementation. During the software unit design and implementation phase, each subsystem is designed and implemented. Reactis Validator can be used to ensure the correctness of any models created during this phase.

Clause 9. Unit testing. The goal of unit testing is to individually test the correctness of each low-level software module. Reactis is extremely useful during this phase, which is covered in more detail in Section 4.5.

Clause 10. Integration testing. The purpose of integration testing is to validate the safety of software systems comprising multiple units. Once again, Reactis will prove to be very useful during this phase. More details on integration test requirements are given in Section 4.6.

Clause 11. Safety requirements verification. Safety requirements verification is performed in order to ensure that the embedded software works correctly in its target environment. Although Reactis is not intended for use in the final target environment, the use of Reactis during the previous phases will minimize the number of issues which arise during target environ-
ment testing. Additionally, tests generated by Reactis during the previous phases can be used to check the embedded software as it executes on the actual target hardware.

The following sections detail how Reactis supports the tasks mandated by clauses 7-11.

4.2. Architectural Design

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.4.1</td>
<td>Use of appropriate notation</td>
</tr>
<tr>
<td>7.4.2</td>
<td>Design considerations</td>
</tr>
<tr>
<td>7.4.3</td>
<td>Modular design</td>
</tr>
<tr>
<td>7.4.4</td>
<td>Identification of software units</td>
</tr>
<tr>
<td>7.4.5</td>
<td>Design aspects</td>
</tr>
<tr>
<td>7.4.6</td>
<td>Component categorization</td>
</tr>
<tr>
<td>7.4.7</td>
<td>New/modified components</td>
</tr>
<tr>
<td>7.4.8</td>
<td>Re-used components</td>
</tr>
<tr>
<td>7.4.9</td>
<td>Allocation of Safety requirements  ✓</td>
</tr>
<tr>
<td>7.4.10</td>
<td>ASIL of combined components</td>
</tr>
<tr>
<td>7.4.11</td>
<td>Software partitioning</td>
</tr>
<tr>
<td>7.4.12</td>
<td>Dependent failure analysis</td>
</tr>
<tr>
<td>7.4.13</td>
<td>Safety mechanisms</td>
</tr>
<tr>
<td>7.4.14</td>
<td>Error detection</td>
</tr>
<tr>
<td>7.4.15</td>
<td>Error handling</td>
</tr>
<tr>
<td>7.4.16</td>
<td>New hazards</td>
</tr>
<tr>
<td>7.4.17</td>
<td>Resource usage</td>
</tr>
<tr>
<td>7.4.18</td>
<td>Architectural design verification   ✓</td>
</tr>
</tbody>
</table>

✓ Supported by Reactis

Figure 8: ISO 26262-6, Clause 7: Architectural design requirements.

The software architectural design phase is covered by Clause 7 of ISO 26262-6. Figure 4.2 lists the requirements of this phase. Requirement 7.4.9 mandates that each safety requirement be allocated to a software component. Reactis Validator can help manage this allocation. When the architectural design is realized as a Simulink model, mapping a safety requirement to a software component means assigning the requirement to a particular Simulink subsystem. When a requirement is formalized as a Validator assertion it must be assigned to a particular subsystem of the model and the desired mapping is maintained by the virtual instrumentation machinery of the tool.

The last requirement (7.4.18) of Clause 7 covers the verification of the design. In a model-based development process, some or all of the software architectural design will be in the form of an executable model. Verification of these models must be done to satisfy ISO 26262-6 requirement 7.4.18.

Figure 9 shows the methods recommended to verify the safety of software architectural designs by Table 6 of ISO 26262-6 (requirement 7.4.18). For designs captured as models (consisting of Simulink, Stateflow, C Code, Embedded MATLAB), Reactis offers a number of capabilities to assist with the verification. For undertaking design walk-throughs and inspections...
Figure 9: Software architectural design verification methods recommended by ISO 26262-6, Table 6 (requirement 7.4.18).

(1a and 1b), Reactis offers:

- Hierarchical browsing of models, including Simulink, Stateflow, C Code (included as S-Functions or Stateflow custom C code), Embedded MATLAB.
- Tracing signals through the model.
- Text search of the model (including C code).

Reactis Validator can be used to test compliance of designs to their software safety requirements. Note that the guided simulation used by Reactis employs a combination of simulation and data- and control-flow analysis (methods 1c, 1f, and 1g).

4.3. Unit Design and Implementation

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.4.1</td>
<td>Safety-related components</td>
</tr>
<tr>
<td>8.4.2</td>
<td>Design notation</td>
</tr>
<tr>
<td>8.4.3</td>
<td>Unit specification</td>
</tr>
<tr>
<td>8.4.4</td>
<td>Design principles</td>
</tr>
<tr>
<td>8.4.5</td>
<td>Unit verification ✓</td>
</tr>
</tbody>
</table>

Figure 10: ISO 26262-6 Software unit design and implementation requirements.

The software unit design and implementation phase is covered by Clause 8 of ISO 26262-6. The requirements which must be satisfied during this phase are listed in Figure 10. Any models created during this phase must be verified in order to satisfy ISO 26262-6 requirement 8.4.5.
Figure 11 shows Table 9 of requirement 8.4.5 which enumerates the methods recommended for verifying the safety of software unit designs and implementations. For unit designs captured as Simulink models, Reactis offers the same capabilities enumerated above for conducting walk-throughs and inspections of architectural designs (1a and 1b). Reactis Validator can also be used in a similar fashion to test compliance of unit designs to their software safety requirements. The underlying guided simulation technology employed to perform the check is a semi-formal verification technique (1c) that employs a combination of data- and control-flow analysis, static code analysis, and semantic code analysis (methods 1e, 1f, 1g, 1h).

<table>
<thead>
<tr>
<th>Method</th>
<th>ASIL</th>
<th>Supported by Reactis?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a Walk-through</td>
<td>++</td>
<td>✓</td>
</tr>
<tr>
<td>1b Inspection</td>
<td>+</td>
<td>✓</td>
</tr>
<tr>
<td>1c Semi-formal verification</td>
<td>+</td>
<td>✓</td>
</tr>
<tr>
<td>1d Formal verification</td>
<td>o</td>
<td>✓</td>
</tr>
<tr>
<td>1e Control flow analysis</td>
<td>+</td>
<td>✓</td>
</tr>
<tr>
<td>1f Data flow analysis</td>
<td>+</td>
<td>✓</td>
</tr>
<tr>
<td>1g Static code analysis</td>
<td>+</td>
<td>✓</td>
</tr>
<tr>
<td>1h Semantic code analysis</td>
<td>+</td>
<td>✓</td>
</tr>
</tbody>
</table>

+ Recommended o No recommendation for or against
++ Strongly recommended ✓ Supported by Reactis

Figure 11: Software unit design verification methods recommended by ISO 26262-6, Table 9 (requirement 8.4.5).

4.4. Testing Methods and Metrics

The testing methods and metrics recommended by ISO 26262-6 are shown in Figure 12. We will show how Reactis can be used to comply with the requirements for unit testing (Section 4.5) and integration testing (Section 4.6).

4.5. Unit Testing

Figure 13 lists the requirements which must be satisfied during software unit testing.

The objective of software unit testing is to demonstrate that all software units conform to their design and do not contain any undesired functionality. A software unit is the smallest testable section of source code. These units are then combined to form a completely functional system. Testing at the unit level prior to testing the system as a whole makes it easier to identify and fix bugs in the code, since the cause of any erroneous behavior found during unit testing must be within the source code unit under test.

Clause 9 of ISO 26262-6 specifies which unit testing methods should be used, how test cases should be derived, and what coverage metrics should be used to assess the effectiveness...
of unit testing.

4.5.1 Unit Testing Methods

The unit-testing methods recommended by Table 10 of ISO 26262-6 (requirement 9.4.3) are shown in Figure 14. There are five methods, which can be explained as follows:

1a. Requirements-based test. Requirements-based testing derives input values from the (previously developed) set of behavioral requirements. Reactis provides two primary ways to capture requirements for use in testing: assertions and user-defined targets. Assertions are used to express properties which should always hold true. For example, the requirement “when the cruise control is engaged, the desired speed and actual speed should never differ by more than 5 miles per hour” could be captured as an assertion. User-defined targets are properties which should hold true at least once, and are typically used to express conditions which should be tested. A typical user-defined target is “the brake should be disengaged for at least 20 consecutive steps during one or more tests.” Section 6 explains in detail how requirements-based testing can be done using Reactis.

1b. Interface test. Interface testing determines if each software unit is properly communicating with the other units in the system. Reactis supports interface testing in several ways. First, Reactis allows you to specify what input values are possible via the use of import constraints. Import constraints are used to restrict the values of an input to a specified set or range. Second, Reactis makes it easy to isolate units for testing via its subsystem extraction feature. This makes
### Requirement | Description
--- | ---
9.4.1 | Safety-related components
9.4.2 | Test planning and execution ✓
9.4.3 | Unit testing methods ✓
9.4.4 | Test cast generation ✓
9.4.5 | Test coverage metrics ✓

✓ Supported by Reactis

#### Figure 13: ISO 26262-6 Software unit testing requirements.

<table>
<thead>
<tr>
<th>Method</th>
<th>ASIL</th>
<th>Supported by Reactis?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements-based test</td>
<td>++</td>
<td>✓</td>
</tr>
<tr>
<td>Interface test</td>
<td>++</td>
<td>✓</td>
</tr>
<tr>
<td>Fault injection test</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Resource usage test</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Back-to-back test</td>
<td>+</td>
<td>✓</td>
</tr>
</tbody>
</table>

+ Recommended ✓ Supported
++ Strongly recommended - Not supported

#### Figure 14: Unit testing methods recommended by ISO 26262-6, Table 10 (requirement 9.4.3).

it possible to test a specific unit of a Simulink model without any manual editing. Third, Reactis for C allows you to create an unlimited number of test harnesses. Each test harness can call any function in a program, allowing separate units to be tested without modifying any source code or even recompiling.

1c. Fault injection test. The goal of fault-injection testing is to ensure the correct behavior of a unit when errors occur. Faults can be injected either at compile-time (typically by mutating the source code in some manner, such as changing a + operator to a − operator), or at runtime (which is typically done by corrupting a value in memory). Reactis does not inject faults in either of these ways. What it does instead is automatically find input values which lead to errors such as overflows, division by zero, etc.

1d. Resource usage test. Resource usage testing attempts to ensure that the unit under test does not consume an excessive amount of CPU time or memory. While resource usage testing is not the primary intended use of Reactis, Reactis for C does report call counts and the maximum recursive depth reached for each C function.

1e. Back-to-back test. The purpose of back-to-back testing is to ensure that a unit under test behaves the same way as the model which served as the design for the unit. Reactis provides very strong support for this type of testing when models are expressed in Simulink/Stateflow and code is developed using the C programming language. Reactis allows you to generate test suites from models and apply them to C source code. Test suites generated from code can also be applied to models if desired. Section 7 explains in detail how back-to-back testing can be done using Reactis.
### 4.5.2 Unit Test Generation

<table>
<thead>
<tr>
<th>Method</th>
<th>ASIL</th>
<th>Supported by Reactis?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a Requirements analysis</td>
<td>++ ++ ++ ++</td>
<td>✓</td>
</tr>
<tr>
<td>1b Equivalence classes</td>
<td>+ ++ ++ ++</td>
<td>✓</td>
</tr>
<tr>
<td>1c Boundary values</td>
<td>+ ++ ++ ++</td>
<td>✓</td>
</tr>
<tr>
<td>1d Error guessing</td>
<td>+ + + +</td>
<td>✓</td>
</tr>
</tbody>
</table>

+ Recommended ✓ Supported
++ Strongly recommended - Not supported

Figure 15: Unit test-case generation methods recommended by ISO 26262-6, Table 11 (requirement 9.4.4).

Figure 4.5.2 lists the methods for deriving unit test cases recommended by Table 11 of ISO 26262-6 (requirement 9.4.4).

1a **Requirements analysis** uses the behavioral requirements of a unit as a basis for determining input values. For example, if a requirement of a function \( f(x) \) is that it return a value of zero when \( x \) is negative, then, based on this requirement a negative value of \( x \), such as \(-1\), would be chosen as one of the test cases. As discussed in Section 4.5.1, Reactis allows requirements to be captured as assertions and user-defined targets. Once this is done, Reactis will automatically find input values which lead to assertion violations and cause targets to be covered.

1b **Equivalence classes** are ranges of input values for which the behavior of a unit under test should be similar. **Equivalence class partitioning** is a form of white-box testing in which the set of possible values for each input is partitioned into subsets based on the behavioral requirements of a unit under test. Once the equivalence classes are determined, a representative value is selected for each equivalence class and a test is generated for the chosen value. The underlying approach of Reactis Tester to generate test suites that maximize different coverage criteria is based on equivalence classes. Here the different equivalence classes are determined by the coverage targets of the model. For each target Reactis selects one test step to represent the equivalence class of all possible steps that might exercise the target.

1c **Boundary values** are the minimum and maximum possible value for an input. For example, if an input \( x \) is an 8-bit signed integer, the maximum value of \( x \) is 127 and the minimum value is \(-128\). Reactis automatically generates test cases for boundary values by analyzing the type of each input.

1d **Error guessing** refers to the use of domain-specific expertise and prior experience to choose input values which are likely to trigger errors. Reactis supports this by allowing input values to be directly specified using its *user-guided simulation* feature.

### 4.5.3 Unit Test Coverage Metrics

Figure 4.5.3 shows the unit test coverage metrics recommended in Table 12 (requirement 9.4.5) of ISO 26262-6. These metrics are defined as follows.
1a **Statement coverage** measures the percentage of statements that have been executed at least once during testing. Reactis supports the metric for both C and Simulink/Stateflow.

1b **Branch coverage** measures the percentage of control-flow branches that have been taken during testing. Reactis directly supports this metric for Simulink/Stateflow. For C code, branch coverage is not directly supported; however, Reactis for C does support decision coverage, which measures the percentage of decisions (a boolean expression used to determine the control-flow of a program) that have been evaluated to true at least once and false at least once. The decision coverage is equivalent to the branch coverage for all the if, while, for, and do...while statements in the unit under test.

1c **Modified Condition/Decision Coverage (MC/DC)** is coverage metric which ensures that each condition in a decision can independently effect the outcome of the decision. The MC/DC standard defines a condition to be a maximal sub-expression of a decision which does not contain any boolean operators. For example, consider the following statement:

\[
if \ ( 50 < x \ \&\& \ x < 100 ) \ y = 0;
\]

This statement contains the decision \( 50 < x \ \&\& \ x < 100 \), which is composed as the logical and of two conditions, \( 50 < x \) and \( x < 100 \). Reactis supports the MC/DC metric for Simulink/Stateflow, Embedded MATLAB, and C code.

### 4.6. Integration Testing

Figure 4.6 lists the requirements which must be satisfied during software integration and testing.

Although unit testing will find many defects in a software system, it is likely that there will be some bugs which arise due to incorrect communications between separate units. The objective of integration testing is to demonstrate that the integrated software units conform to the architectural design, including safety properties. ISO 26262 specifies which integration test methods should be used, how test cases should be derived, and what coverage metrics should be collected in order to judge the effectiveness of the integration testing.

The software integration test-methods recommended by Table 13 of ISO 26262-6 (requirement 10.4.3) are identical to ones recommended for unit testing; therefore, Reactis can be used at the integration level in the same way it can at the unit level as described in Section 4.5.1.
Similarly the test generation methods recommended by Table 14 of ISO 26262-6 (require-
ment 10.4.4) are identical to the methods for unit test generation, so discussion of unit test
generation with Reactis applies for the generation of tests for integration testing as well. The
integration test metrics recommended by ISO are listed in Figure 4.6.

<table>
<thead>
<tr>
<th>Metric</th>
<th>ASIL</th>
<th>Supported by Reactis?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>1a</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>1b</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

+ Recommended ✓ Supported
++ Strongly recommended

Figure 18: Integration test coverage metrics recommended by ISO 26262-6, Table 15 (require-
ment 10.4.6).

4.7. Verification of Software Safety Requirements

The last phase of software verification under ISO 26262 is the verification of software safety
requirements phase, during which the embedded software is tested to ensure that it operates
safely in the target operating environment. Figure 4.7 lists the requirements which must be satisfied during software safety verification. Since the testing in this phase should be performed on the actual target hardware, the best use of Reactis is to generate tests from the model and/or C code which are then exported from Reactis and used to test the embedded software in its target environment. Test data can also be imported from the target environment into Reactis and compared against the model and/or C code. Details on how to do this are given in Section 7.4.

5. An ISO 26262 Process using Reactis

In this section we describe how the specific ISO 26262 tasks enumerated in the previous section can be assembled into a complete ISO 26262-compliant model-based process (shown in Figure 20).

Figure 20: An ISO 26262 model-based development process. Numbers indicate the relevant clause of ISO 26262-6.

In the reference process, executable models are produced as components of the architectural and unit designs, which are used to guide the implementation of the software. During the testing phases, the conformance of the implementation is confirmed via testing. Reactis can help during all of these phases.
During the architectural design phase (ISO 26262-6 clause 7), Reactis can be used during walk-throughs, inspections, and to verify architectural designs by testing them against their safety requirements (ISO 26262-6 requirement 7.4.18). The architectural design phase is described in Section 4.2. Requirements-based testing of models with Reactis is explained in Section 6.

During the unit design and implementation phase (ISO 26262-6 clause 8), Reactis can be used for walk-throughs and inspections and to verify unit designs by testing them against their safety requirements. When appropriate Reactis can also help compare the behavior of units to subsystems of architectural models (ISO 26262-6 requirement 8.4.5). See Section 6 for more details. Also see section 4.3 for some additional details of the unit design and implementation phase.

During the unit testing phase (ISO 26262-6 clause 9), Reactis can be used to perform back-to-back comparisons of implementations and models, and also directly test C code against requirements if desired, in order to satisfy ISO 26262-6 requirement 9.4.3 (see item 1e in Figure 14). Additional details of the unit testing phase are given in Sections 4.5-4.5.3, and guidelines for performing back-to-back testing with Reactis are given in Section 7.

During the integration testing phase (ISO 26262-6 clause 10), Reactis can be used to perform a second round of back-to-back testing, this time comparing systems composed from multiple units against architectural models, in order to comply with ISO 26262 requirement 10.4.3. See Section 7 for details on how such testing is done.

During the safety requirements verification phase (ISO 26262-6 clause 11), Tests generated by Reactis can be used to test the embedded software in its target environment. More details on this are given in Section 4.7.

6. Requirements-Based Testing with Reactis

![Workflow for testing model requirements with Reactis.](image)

Models can serve as an essential part of the architectural or unit-level design of a system. When models are used in an architectural design, ISO 26262-6 requirement 7.4.18 requires verification to be done. Similarly, when models are used in a unit-level design, ISO 26262-6 requirement 8.4.5 requires verification of the models. During the verification process, each model will be tested against safety requirements which were identified during a previous
development phase (safety requirements specification). Reactis Validator can be used to perform requirements-based testing on Simulink/Stateflow models.

Before testing can be done, a model must first be annotated with the relevant safety requirements, as shown in Figure 21. This is done by loading the model in Reactis, navigating to the appropriate subsystem within the hierarchy of the model, and then right-clicking in the main panel where you want to place a Validator objective. A pop-up menu lets you choose the type of objective to add (assertion, user-defined target, or virtual source). After choosing the objective type, an editor for the new objective will appear.

![Figure 22: Creating an expression assertion in Reactis.](image)

For example, a cruise control system would typically be required to disengage during braking. This requirement can be represented by an assertion that it is never the case that the brake signal and the cruise control’s active signal are ever true at the same time. Figure 22 shows such an assertion being created in Reactis.

Each Validator objective you add will appear as part of the model schematic in Reactis. Recall that, as noted in Section 3.3, the objectives only appear within Reactis and are stored in a separate file, so that the model itself is never modified. Figure 23 shows an annotated cruise control model loaded in Reactis, to which four objectives have been added (these are shown in the magnified portion of the screen). The brake assertion whose creation is shown in Figure 22 is represented by the lightning bolt in the lower left corner of the magnified region.

Once the model has been annotated with Validator objectives, a check for violations of any requirements can be launched from the Reactis Validator menu entry “Check Assertions”, or by launching Reactis Tester. Reactis will automatically generate tests based on the assertions and the model. These tests will include any tests that lead to an assertion violation or cover
a user-defined target. In addition to Validator objectives, the tests will also aim to maximize the number of built-in targets (such as MC/DC) exercised. A test suite with a high level of MC/DC coverage and no assertion violations gives assurance that the model is conforming to the checked requirements.

Depending on the test parameters and model size, generating a test suite can take anywhere from a few seconds to multiple hours. As testing proceeds, a progress dialog shows the current coverage status for each coverage objective and estimated time remaining.

During test suite generation, Reactis may detect runtime errors or assertion violations, in which case it will remember the inputs which lead to the erroneous behavior. When this happens, you can use Reactis Simulator to debug your model and determine the cause of the error. The result of executing a test suite is shown in Figure 23. Note that the Speed assertion is colored yellow, which indicates that an assertion violation occurred. When this assertion is hovered over, Reactis displays blue lines showing the inputs to the assertion and text giving the test and step number when the violation occurred. As shown in Figure 23, the inputs to the assertion are the signals active and speed, and the first violation occurred at step 962 of test 29.


When using model-based development within an ISO 26262 process, ISO 26262-6 requirements 9.4.3 and 10.4.3 recommend performing back-to-back comparisons of a model and the implementation derived from the model. The goal of back-to-back testing is to determine that an implementation and model both produce the same outputs when given the same inputs. There are four essential requirements for back-to-back testing to be successful. First, there should be a high degree of confidence in the correctness of the model due to prior testing
of the model against its requirements. Second, the implementation should produce outputs which are reasonably close (small differences are likely to rounding of results during numerical calculations) to the outputs of the model for all inputs. Third, the tests which were used to perform the comparison should achieve a high degree of coverage of the model and its requirements. Fourth, the tests should achieve a high degree of coverage of the implementation. This last point deserves some emphasis because, if you simply take the tests which were used during model verification and apply them to the implementation, you may satisfy the first three criteria without covering the full implementation structure.

Figure 24: Back-to-back comparison of a model and code.

A generic back-to-back comparison process is shown in Figure 24. For the sake of brevity, the implementation is referred to simply as code. As shown in Figure 24, there are four fundamental activities involved in back-to-back testing. These are (1) generating tests from the model, (2) executing tests on the code, (3) generating tests from the code, and (4) executing tests on the model.

There are four basic ways to perform such testing with Reactis, depending primarily on how the code is to be executed. Code can be tested using Reactis for C, the Reactis for C Plugin, the Reactis runtests utility, or an external tool. Each of these approaches has its own unique advantages and disadvantages, which need to be considered before deciding which approach to use.

7.1. Back-to-Back Testing with Reactis for C

If the implementation is written in C, Reactis for C can be used, as shown in Figure 25. The transitions in Figure 25 are numbered 1-5. The numbers correspond to the five steps in the testing process:

1. **Generate a test suite from the model.** Reactis Tester is used to generate the test suite from the model, as described in Section 6. The test suite will be in the form of a Reactis test suite (.rst) file.

2. **Load the C code in Reactis for C.** The C code must be loaded in Reactis for C before it can be tested. Reactis for C uses its own internal C compiler, so it must know where each source file is located. Reactis stores this information in a special file with extension .ram. A graphical editor eases the creation and editing of .ram files.
Additionally, you will need to define at least one test harness. A test harness specifies a top-level entry function to call, how to pass test inputs into the code under test, and how to read test outputs from the code. Reactis for C includes a graphical editor for creating test harnesses, which are stored in files with extension .rsh.

3. Test the C code. Once the program is loaded in Reactis for C, testing is ready to proceed. The test suite produced from step 1 is loaded into Reactis Simulator and executed using the code and test harness from step 2. Any output differences or runtime errors which occur as the test suite is executed will be automatically flagged by Reactis. The debugging features of Reactis Simulator can then be used to determine the cause of any deviations.

4. Generate a test suite from the C code. In this step, Reactis Tester is used to create a test suite which maximizes structural coverage of the implementation under test. The test harness from step 2 will be used, so no additional work is required. Optionally, Reactis Validator can be used to generate additional tests based on assertions and user-defined targets which have been added to the C code.

5. Test the model. In this step, the model is executed using tests generated from the implementation. The test suite produced from step 4 is loaded into Reactis Simulator and executed using the model from step 1. Reactis will automatically flag any output differences or runtime errors, which can then be debugged with Reactis Simulator.

The advantages of using Reactis for C are that (1) the test suites (.rst files) produced by Reactis from the Simulink model can be directly re-used in Reactis for C, (2) Reactis for C will automatically detect many types of runtime errors (such as memory errors) while executing tests, (3) Reactis Simulator can be used to debug any errors that occur, and (4) Reactis for C can be used to generate tests from the implementation. The disadvantages of the approach are (1) there will be some effort involved in creating a test harness in Reactis for C, and (2) the
Reactis for C execution environment will likely differ to some degree from the target execution environment.

7.2. Back-to-Back Testing with the Reactis for C Plugin

Figure 26: Using the Reactis for C Plugin to perform back-to-back testing.

1. **Generate a test suite from the model.** Reactis Tester is used to generate the test suite from the model, as described in Section 6. The test suite will be in the form of a Reactis test suite (.rst) file.

2. **Load the C code in Reactis for Simulink.** The C code must be loaded in Reactis for Simulink before it can be tested. First, you will need to create a “wrapper” model and S-function which has the same top-level input and output ports as the model. If you are using an automated code-generation tool to create the C code, your tool likely can create the wrapper model automatically. Finally, in order to test the C code within the S-function, you will need to create a .rsm file (see step 2 of Section 7.1).

3. **Test the C code.** Once the wrapper model for the implementation is loaded in Reactis, testing is ready to proceed. The test suite produced from step 1 is loaded into Reactis Simulator and executed. Any output differences or runtime errors which occur as the test suite is executed will be automatically flagged by Reactis. The debugging features of Reactis Simulator can then be used to determine the cause of any deviations.

4. **Generate a test suite from the C code.** In this step, Reactis Tester is used to create a test suite which maximizes structural coverage of the S-function source code, which includes all
of the implementation C code. Optionally, Reactis Validator can be used to generate additional tests based on assertions and user-defined targets which have been added to the C code or the top-level of the wrapper model.

5. **Test the model.** In this step, the model is executed using tests generated from the implementation. The test suite produced from step 4 is loaded into Reactis Simulator and executed using the model from step 1. Reactis will automatically flag any output differences or runtime errors, which can then be debugged with Reactis Simulator.

The advantages of this approach include all the advantages of using Reactis for C standalone. One additional advantage is that no test harness will be required to test the S-function. The downside is that there will be work required to create S-function wrapper code for the implementation, unless you are using an automated code generator, in which case the code generator will probably produce S-function wrapper code.

### 7.3. Back-to-Back Testing with Reactis Runtests

A third approach which also requires the implementation to be in the form of Simulink S-function is to test the implementation within MATLAB using the *runtests* command (included in the Reactis distribution), as shown in Figure 27. The testing process depicted in Figure 27 consists of four steps:

1. **Generate a test suite from the model.** Reactis Tester is used to generate the test suite from the model, as described in Section 6. The test suite will be in the form of a Reactis test suite (.rst) file.

2. **Export the test data to a .m or .mat file.** In this step, the test data is exported to a format which can be processed within MATLAB. This is done by loading the model under test and .rst file in Reactis and then using Reactis Simulator’s test suite export feature. The test data can be exported as either a MATLAB script (.m) or MATLAB binary data (.mat). The only user input required is to select the format of the exported test data and name of the file which will contain the exported data.
3. **Load the code in MATLAB.** The implementation code must be loaded in MATLAB before it can be tested. First, you will need to create a “wrapper” model and S-function which has the same top-level input and output ports as the model. If you are using an automated code-generation tool to create the C code, your tool likely can create the wrapper model automatically. The S-function will need to be compiled using MATLAB’s `mex` command.

4. **Test the C code.** Once the wrapper model for the implementation is loaded in MATLAB, testing is ready to proceed. Reactis includes a MATLAB script called `runtests` which automates the testing of MATLAB models. The model from step 3 is executing using the test data contained in the `.mat` file produced during step 2. Any output differences which occur as the test suite is executed are flagged for investigation. Since the S-function in is the form of binary code, direct debugging of output differences may prove difficult. A better option is to use the Reactis for C Plugin as a debugging tool.

The advantage of the `runtests` approach is that the S-function will execute faster, because MATLAB compiles S-functions into native machine code which is then directly executed by the CPU, while the Reactis for C Plugin generates virtual machine code which is necessarily slower. There are two primary disadvantages. First, the errors which would trigger an immediate error in the Reactis for C Plugin will typically not have any immediate effect in the compiled S-function. Instead such errors are likely to produce corrupt values in the program leading to an eventual crash or invalid output which is much more difficult to debug. One option for debugging such cases is the Reactis for C Plugin. Second, Reactis will not be able to generate any tests from, or measure coverage of the compiled S-function. Hence, there is the possibility of latent code defects which are not being reached during testing.

### 7.4. Back-to-Back Testing with External Tools

![Figure 28: Exporting a Reactis test suite for use with an external tool.](image)

If the implementation is coded in a language other than C and is not in the form of an S-function, or you want to do testing using a specialized target environment, then Reactis can export the test suite to comma-separated value (`.csv`) format for use by an external tool. This approach is shown in Figure 28, and consists of three steps (labeled 1-3 in Figure 28):
1. **Generate a test suite from the model.** Reactis Tester is used to generate the test suite from the model. The test suite will be in the form of a Reactis test suite (.rst) file.

2. **Export the test data to a .csv file.** In this step, the data in the .rst file created during step 1 is loaded into Reactis Simulator and then exported to a comma-separated value (.csv) file.

3. **Execute tests using external tool.** In this step, the external tool executes the implementation code using the test data produced in step 2. The external tool will be responsible for comparing code outputs to model outputs and reporting any differences. Alternatively, the outputs of the external tool can be saved and Reactis can be used to check for output differences, as shown in Figure 29.

![Figure 29: Importing test data from an external tool for use with Reactis.](image)

Figure 29 shows how test data produced by an external tool can be used within Reactis. As indicated by the numbers in Figure 29, there are four steps required to do this:

1. **Generate a test suite from the code.** The external testing tool is used to generate a test suite from the implementation code.

2. **Export the test data to a .csv file.** The test data generated by the external tool is converted to comma-separated value format and stored in a (.csv) file.

3. **Import test data.** The .csv test data produced during step 2 is imported into Reactis. Reactis includes a test data import utility that lets you easily map test data onto model inputs and outputs. You will need to make sure that each column in the .csv file is mapped to the appropriate model component.

4. **Execute tests with model.** At the click of a button, Reactis Simulator will execute all tests in the imported test suite with the model. Any runtime errors or output differences which occur will be flagged. Additionally, you can visually inspect the model after all tests have been executed to see which targets in the model were covered, or generate a coverage status report.

The advantage of using an external tool is the very high degree of flexibility in testing it allows, such as the ability to perform testing using specialized hardware. Testing can be done at a level which is close to the final target on which the software will be deployed.
A minor disadvantage of using an external implementation test tool is the extra work required to exchange test data between Reactis and the external tool. Additionally, debugging any errors or output differences which are detected may be more difficult, although it might be possible to use Reactis Simulator as a debugging aid.

7.5. Errors During Testing

It is quite possible that one or more tests will fail to complete due to a runtime error such as a memory access error or divide by zero. In this case, Reactis Simulator will prove very helpful in determining the source of the error. Simulator has a number of features which make it easy to inspect an execution sequence that leads to an error, including reverse-stepping, value inspection, and scopes. These features are shown in Figure 6.

7.6. Comparing Outputs

Those tests which do not fail outright will need to be checked to ensure that the model and implementation produce outputs which are reasonably close. Note that there are a number of legitimate reasons why the outputs will not be exactly the same. First and foremost, since digital computations are done using finite precision, the final result of any calculation depends on the order in which the calculation is done. In other words, the standard algebraic laws of arithmetic do not apply: it will not be the case that \( x \cdot (y + z) = x \cdot y + x \cdot z \) for all values of \( x \), \( y \), and \( z \). Second, it is possible that, in order to improve performance, the implementation may use different numerical types than the model. A model might perform all calculations using double precision floating-point values, while an implementation based on the model might use a 32-bit fixed-point representation to perform the same calculations.

For these reasons, there will usually be some deviation between the outputs of the model and code. For this reason, Reactis allows you to specify a tolerance for each output. When comparing two results, the comparison is considered successful if the difference is within the allowed tolerance.

7.7. Evaluating Coverage Results

Once all errors and output differences have been fixed, the issue of the structural coverage of the implementation remains as the last hurdle to be cleared in order to have a strong level of confidence that the model and implementation behave the same for all inputs. Reactis Simulator lets you visually inspect the coverage of a model and/or C code. The coverage status of every target is displayed graphically, including MC/DC targets. For example, in Figure 6, targets which have not been covered are colored red, covered targets are colored black, and the state at which execution is paused is colored green. Reactis also generates comprehensive coverage reports which give details on the status of all targets. This information allows you to identify model/code regions which have low coverage, and focus additional testing accordingly.

Reactis also provides several ways to improve coverage. The first is the Reactis Tester launch dialog, which lets you specify how long Tester will have to generate tests. The second is the ability to add user-defined inputs to any test suite. The third are Validator objectives,
which can be used to do things such as hold an input signal at a particular value for a specified amount of time. These features work in synergy with Reactis’ patented guided simulation algorithm, which takes a single test step that leads to a new target being covered and, based on that step, finds additional steps which lead to the coverage of nearby targets.

8. Conclusions

For organizations which are using Simulink/Stateflow and C to develop software using model-based design, Reactis provides excellent support for the validation and testing activities required by ISO 26262-6. In particular, Reactis does the following:

- Supports the coverage metrics recommended by ISO 26262, including the most stringent (MC/DC).
- Automatically generates compact test suites which maximize coverage.
- Performs back-to-back comparison of C code and Simulink/Stateflow models.
- Supports functional requirements-based testing via Validator assertions and user-defined targets.
- Detects a variety of runtime errors during the testing process, including memory errors. Furthermore, this detection occurs immediately, not at some point in the future when the memory error triggers an observable malfunction.
- When errors are found, produces a specific sequence of inputs which leads to the error. Reverse single-stepping with value inspection helps determine the defect in the source code which caused the error.
- Presents coverage data both textually (in the form of a customizable report that can be incorporated into work products required by ISO 26262), and graphically (via a browser which allows you to examine a Simulink/Stateflow model or C code and quickly identify which parts of a system require additional testing).

Please see the Reactive Systems web site at www.reactive-systems.com for ordering information and for instructions on how to download a free 30-day evaluation copy of the software.