Test Case Generator User’s Manual

Version 1.1

Mike Whalen, Rockwell Collins\University of Minnesota

This research was developed with funding from the Defense Advanced Research Projects Agency.

The views, opinions and/or findings expressed in this press release should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.



Table of Contents

[1 Introduction 1](#_Toc454263026)

[2 Installation 5](#_Toc454263027)

[3 Invoking the Test Case Generator 6](#_Toc454263028)

[4 Viewing the Output of the Test Case Generator 8](#_Toc454263029)

[4.1 Viewing Test Cases 8](#_Toc454263030)

[4.1.1 In the Console 8](#_Toc454263031)

[4.1.2 In a Tabular View 9](#_Toc454263032)

[4.1.3 In a Spreadsheet 10](#_Toc454263033)

[4.2 Simulating Test Cases 11](#_Toc454263034)

[4.3 Inductive Test Cases 12](#_Toc454263035)

[4.4 Navigating to Related Contract Element 13](#_Toc454263036)

[5 Extending and Persisting Test Suites 14](#_Toc454263037)

[5.1 Persisting / Restoring Test Suites 14](#_Toc454263038)

[5.2 Extending the Currently Loaded Test Suite with Simulation Results 15](#_Toc454263039)

[6 Test Generator Preferences 17](#_Toc454263040)

|  |  |  |  |
| --- | --- | --- | --- |
| Release History | | | |
| Date | Version | Author | Description |
| June 21, 2016 | 1.0 |  | Initial Release |
|  |  |  |  |
|  |  |  |  |

# Introduction

Software is ubiquitous in safety-critical systems, which have the potential to cause loss of life, injury, or other serious damage to property and environment. The size and complexity of this software continues to grow, making it ever more difficult to capture the correct requirements, design the software correctly, and verify to a high level of confidence that we have the right requirements and that the software indeed satisfies those requirements. To make design and construction possible, such a complex system is typically organized as a composition of subsystems that can themselves be further decomposed if necessary. This hierarchical aspect of design is critically important; it allows the complexity of the entire system to be managed through partitioning and abstraction.

This decomposition touches both requirements and architecture, since the architecture describes the structure of the decomposition, and this will affect how requirements “flow down” to each subsystem. We believe that requirements should be organized into hierarchies that follow the architectural decomposition of the system because the act of decomposing a system into components induces a requirements analysis effort in which we need to ascertain whether the requirements allocated to subcomponents in the architecture are sufficient to establish the system-level requirements. We also need to determine whether any assumptions on a component’s environment made when allocating requirements to that component can be established.

To convincingly argue that a system has the desired effect in its environment (the system satisfies its requirements), Hammond et al. developed the notion of a Satisfaction Argument, based on Jackson and Zave’s World and the Machine model. This approach attempts to establish that system requirements hold through an argument involving (i) the specification of the system behavior and (ii) assumptions about the domain of the system.

To formalize satisfaction arguments, assume-guarantee contracts provide an appropriate mechanism for capturing the information needed from other modeling domains to reason about system-level properties. In this formulation, guarantees correspond to component requirements, and assumptions correspond to the environmental constraints that are used in verifying the component requirements. For formally verified components, assumptions are assertions or invariants on component inputs that are used in the proof process. A contract specifies precisely the information that is needed to reason about the component’s interaction with other parts of the system. Furthermore, the contract mechanism supports a hierarchical decomposition of the verification process that follows the natural hierarchy in the system model.

Rockwell Collins and the University of Minnesota have constructed the *AGREE* tool suite to reason formally about such decompositions. In this approach, we use the AADL architecture description language to describe the decomposition of the system architecture along with an *AGREE Contract annex*, which extends the AADL language in order to describe formal requirements.

Given AADL models that use the AGREE requirements annex, we have a suite of tools that can simulate models, prove properties, and generate test cases from agree contracts. The proof process involves a series of *satisfaction arguments* that each demonstrate, for one “layer” of the architecture at a time, that the requirements in a system level contract of this layer are proved given the requirements allocated to the immediate subcomponents of the layer.

This document describes a test-case generation tool that was added to the AGREE tool suite. Using this tool, it is possible to generate rigorous test suites *directly from requirements specifications*. The test suites generated by the tool satisfy a rigorous metric called *Unique First Cause (UFC)*, which is similar to the *Modified Condition/Decision Coverage (MCDC)* metric used for testing critical avionics systems. In addition, it is possible to save/replay test suites and to record simulation results into a test suite.

Unlike compositional verification that verifies a model for all possible combinations of inputs and reachable states, even rigorous testing can only verify a tiny fraction of the possible behaviors of an AADL model, so an obvious question is: why bother generating tests from requirements? The answer is three-fold

1. The tests check for *vacuity* of properties within the model
2. The tests allow the system engineer to quickly explore different “interesting” scenarios within the model.
3. The tests can be used to check an *implementation* for conformance to the contract.

Each of these uses will be examined briefly here.

**Vacuity:** When using formal verification at scale, it is possible to make mistakes in formalizing contracts, just like any other development activity. Usually, such mistakes lead to *counterexamples,* which describe proof failures because some aspect of the model was incorrectly specified. However, occasionally, these mistakes make proofs *vacuously true*, which means that the requirement of interest was proved, but this proof in fact hides a mistake in the model.

Several years of experience in formal verification of hardware at IBM have demonstrated that during the first formal verification runs of a new hardware design, typically 20% of formulas are found to be trivially valid, and that trivial validity always points to a real problem in either the design or its specification or environment. At Rockwell Collins, a smaller but still substantial number of trivially valid properties have been discovered when reasoning about avionics software. *Trivial validity* in this case means that some part of the requirement is *irrelevant* to the proof (we could remove it and still derive a proof). For example, consider the implication:

A → (B > 0)

If A is always false, then B is unnecessary, and we could simplify the property to

¬A

Similarly, if (B > 0) is always true, then A is unnecessary, so we could simplify the property to:

(B > 0)

The UFC test metric generates tests that demonstrate the *independent effect* of each *condition* in the property, where a condition is a Boolean expression with no operators (in the example above, the conditions are {A, B > 0}). In other words, the test is an explanation of why the condition is necessary in order to prove the property. On the other hand, if there is no test that demonstrates the independent effect of the condition, then this condition is *vacuous.* So, by running the test case generator, we both generate *witnesses* (test cases) that demonstrate why conditions are necessary, and in the case when it is not possible to generate tests, we demonstrate that formulas are *vacuous* – the failing test case describes the condition that is unnecessary to the satisfaction of the property.

**Exploration**

The scenarios that are generated by the test case generator illustrate interesting behaviors of the model. They describe, for each condition, a trace that illustrates why that condition is necessary for the proof. Test cases generated for an AADL model can be run through the simulator to discover many discrepancies between the contract and its component contracts before investing time and effort in compositional verification.

**Conformance**

Automatically generating tests for a legacy or vendor component provides a straightforward way for the supplier to test if the physical component actually satisfies its contract. Finally, it reduces the overall cost of verification by eliminating the cost of writing test cases by hand.

**Overview of How Tests are Constructed**

Metric-based automated test generation consists of three steps. First, a coverage criterion is developed that defines when a set of tests provide sufficient coverage of the unit under test. Second, a test generation strategy is defined to generate tests that achieve that level of coverage. Third, a test refinement strategy is developed to optimize the set of tests created. The purpose of a coverage criterion is to ensure that the generated tests provide effective fault detection.

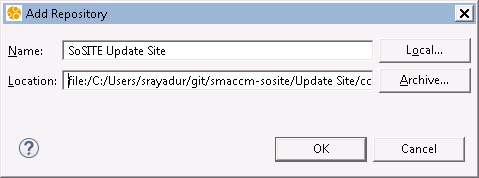
The Unique-First-Cause (UFC) criterion provides effective fault detection of assume/guarantee contracts. Applying the coverage criterion to a specific assume/guarantee contract generates a number of predicates that must be satisfied to achieve the desired coverage of the contract. The essence of the test generation strategy is to assert, for each such predicate, that there is no possible sequence of input values satisfying the predicate. These assertions are then given to a model checker along with the formal specification of the system to generate counterexamples yielding a set of inputs do indeed satisfies the predicate.

Instructions for installing and using the test case generator are provided in this User’s Manual.

# Installation

The test case generator can be installed from the Eclipse SoSITE Update Site for Eclipse. It is assumed that the user has a working installation of OSATE 2 (version 2.1.x) for editing AADL models.

1. Production version of OSATE2 can be obtained and installed following the instructions at: <https://wiki.sei.cmu.edu/aadl/index.php/OSATE_2_download_page>
2. To set-up SoSITE update-site: Start OSATE and select Help -> Install New Software and setup the SoSITE Update Site as a local repository

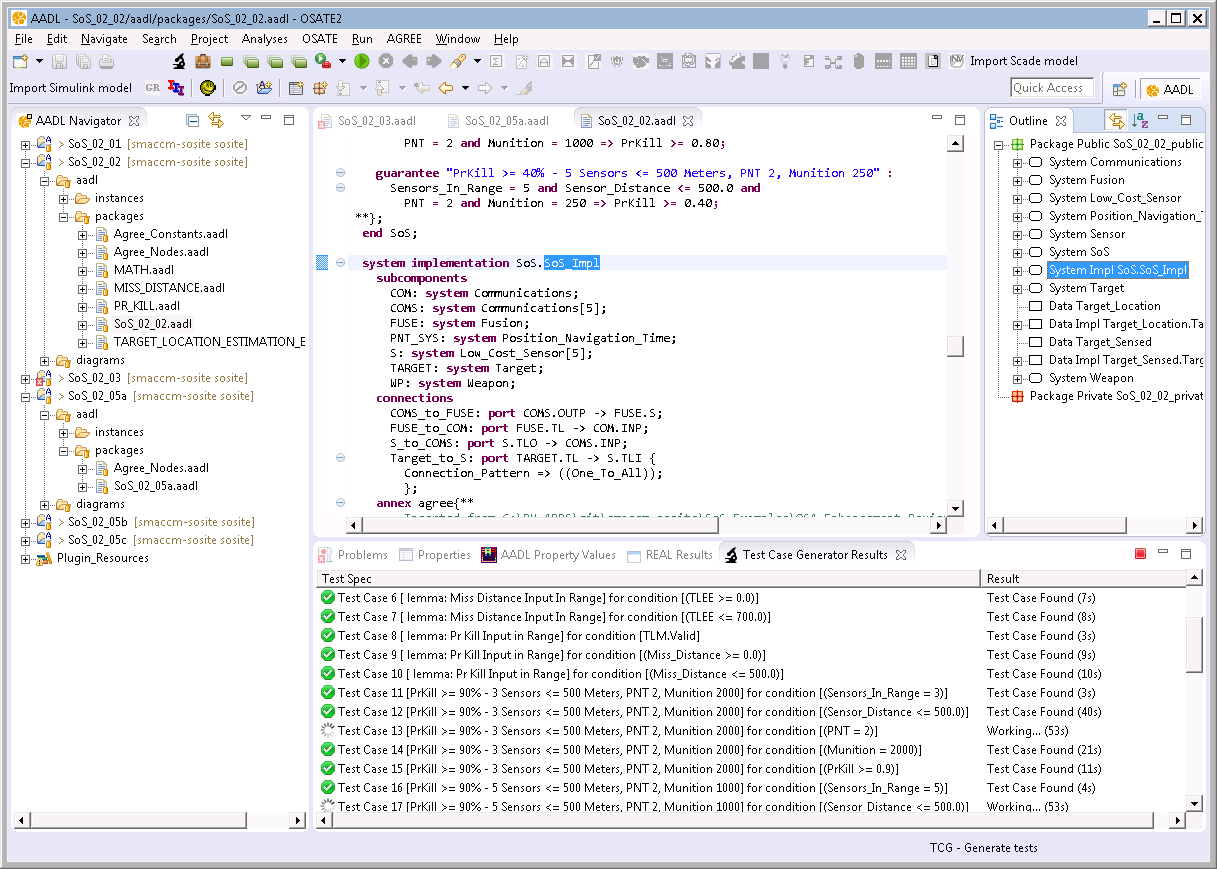


1. Uncheck "Group Items by Category", Select Test Case Generator and Simulator components, select Next, Accept the license terms and select Finish.
2. Restart OSATE2 to activate the Test Case Generator.

# Invoking the Test Case Generator

The Unique First Cause test cases generator can be invoked from the OSATE User Interface.

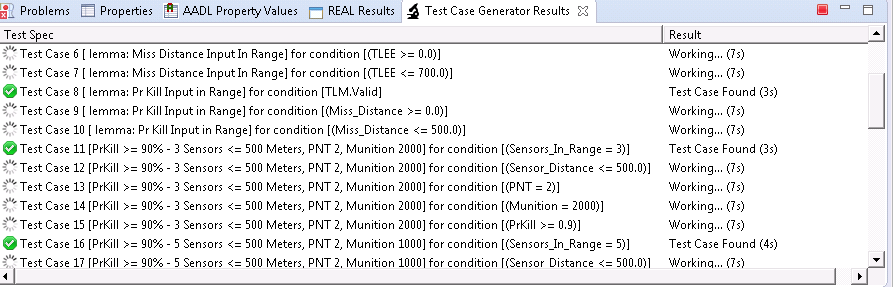
1. Select the *System Implementation* from the Outline view for your AADL model. E.g.:



1. Either from the main menu item “AGREE” or from the context menu for *System Implementation* (activated on right-click), select “Run UFC test case generator”:

|  |  |
| --- | --- |
|  |  |

This invokes the test case generator which constructs UFC test specifications for all the conditions in the *assumptions* and *guarantees* specified the AGREE annex of the AADL model. The “Test Case Generator Results” pane shows the generation status:



# Viewing the Output of the Test Case Generator

The “Test Case Generator Results” pane shows the output produced by the TCG. The “Test Spec” column shows a descriptive text for each generated test specification. The specification shows a test case number, the descriptive name of the contract element and the specific Boolean condition to be exercised by the test case. E.g.,



As the test specifications are processed by the TCG, the processing status of each test specification is shown as an icon prefixing the specification as well as a short text in the “Result” column.

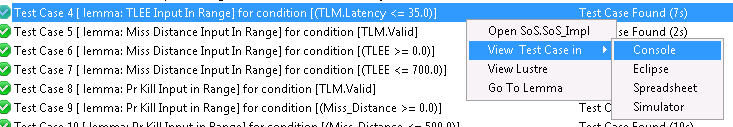
|  |  |  |
| --- | --- | --- |
| **Icon** | **Result** | **Description** |
| C:\Users\srayadur\git\smaccm-sosite\fm-workbench\tcg\com.rockwellcollins.atc.tcg\icons\waiting.png | Working…[true for ***n*** steps] (***Τ***s) | TCG is still processing this test specification. No test case of ***n*** steps or less exists. |
| C:\Users\srayadur\git\smaccm-sosite\fm-workbench\tcg\com.rockwellcollins.atc.tcg\icons\tick_octagon.png | Valid (***Τ***s) | A valid test case has been found for this test specification. |
| C:\Users\srayadur\git\smaccm-sosite\fm-workbench\tcg\com.rockwellcollins.atc.tcg\icons\red_delete.png | Invalid (***Τ***s) | No test case exists for this test specification. Indicates potential vacuityin the contract. |
| C:\Users\srayadur\git\smaccm-sosite\fm-workbench\tcg\com.rockwellcollins.atc.tcg\icons\orange_qmark.png | Unknown [true for ***n*** steps] (***Τ***s) | No test case test case has been found within the bounds set for test length and processing time. |
| (***Τ*** *in seconds, is the time taken to process the spec*) | | |

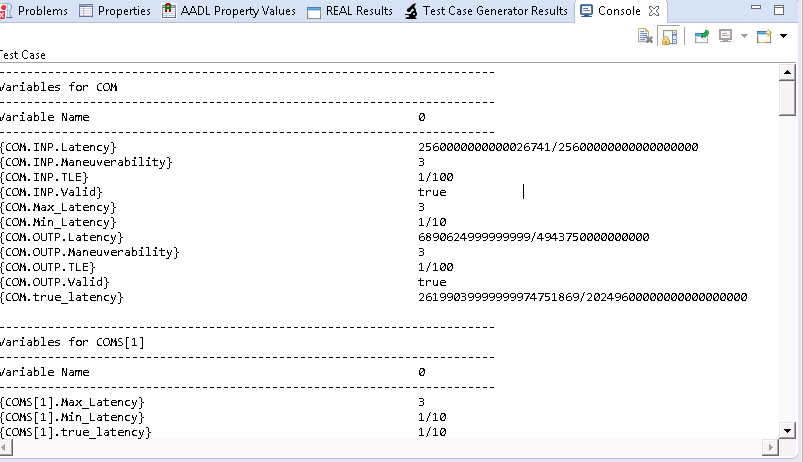
## Viewing Test Cases

The context-menu (activated by right-click) for the test specifications listed in the output pane “Test Case Generator Results” provides multiple ways of viewing a generated test case.

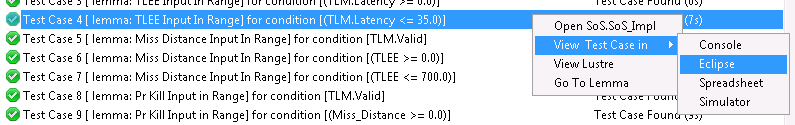
The test cases are to be understood as a sequence of steps with values for all the variables in each component of the system.

### In the Console

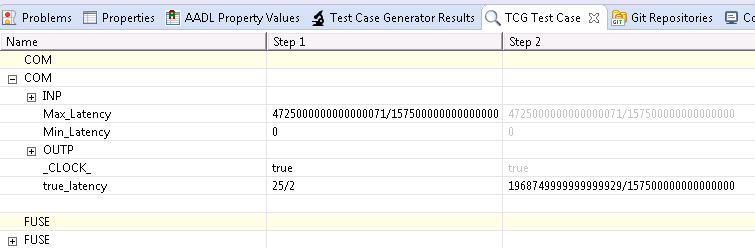


This view is useful for obtaining a simple textual representation of the generated test case. The textual representation (or parts of it) can be easily copied and pasted into other files and documents as needed. 

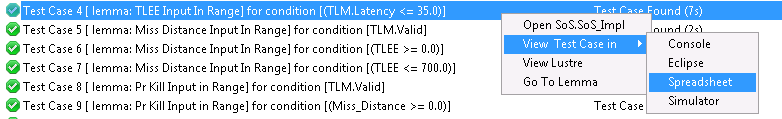
### In a Tabular View



This provides a more structured view of the test case as a table. Variables are hierarchically organized component-wise on the architectural lines and each step of the test case is shown as a separate column. Values that do not change at a step are grayed for visual distinction.

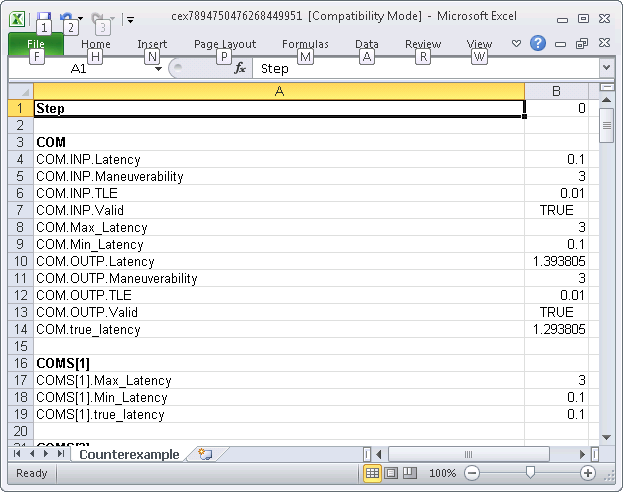


### In a Spreadsheet

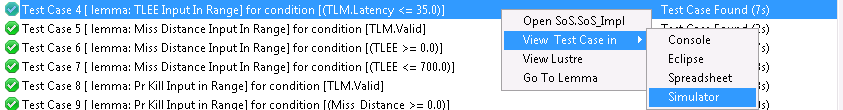


The tabular form of the test case lends itself to be easily exported to a Spreadsheet format such as Microsoft Excel. This is particularly useful if the test case is to be processed further by external programs that handle input files in formats such as comma-separated values.

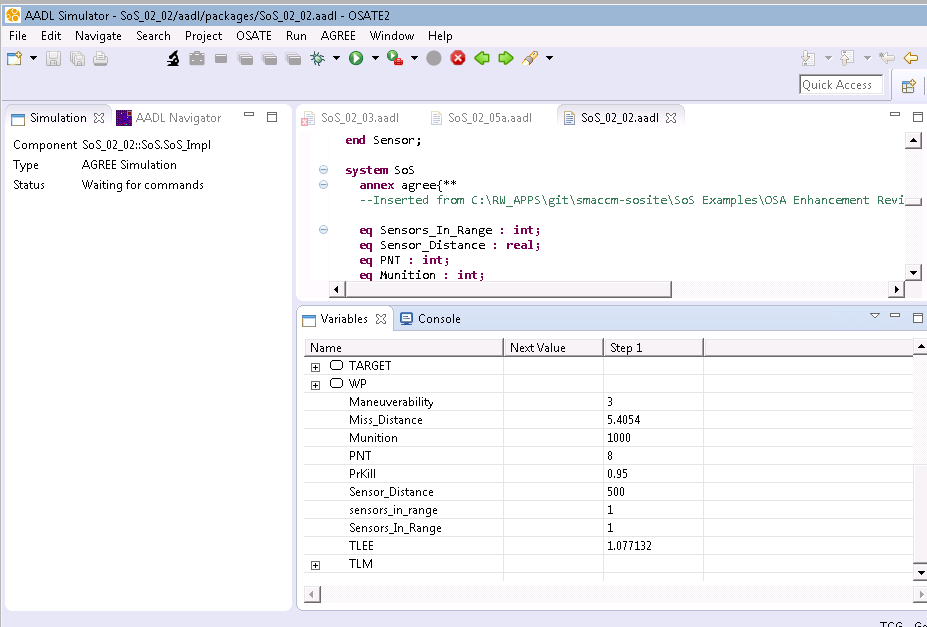
Following is an exported view of the same test case in Microsoft Excel:



## Simulating Test Cases

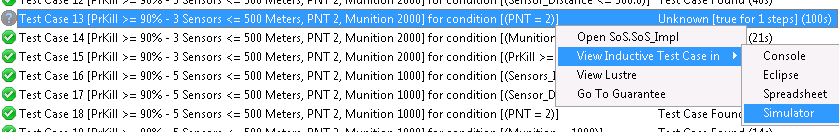
The generated test cases can not only be viewed in various formats statically but can also be viewed dynamically by animating its execution. This is enabled by simply selecting to view the test case in the simulator: 

This will load the test case in the simulator and bring up the simulator view for the AADL model. This provides a powerful way to step through the execution of the test case forwards and backwards, and further extend the test case using the simulator’s functionality. An execution trace of the system constructed or modified by the simulator can in turn be exported and saved as test cases.



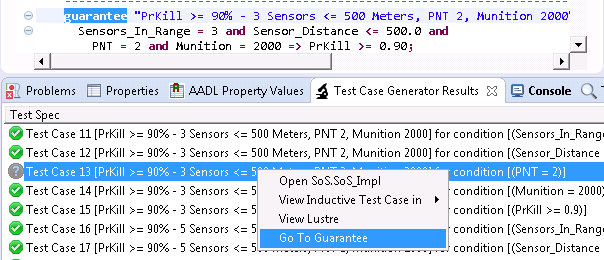
## Inductive Test Cases

If a test obligation status is Unknown at the end of test generation, then it means that the generator was unable to find a test case within the allotted time budget. In such cases an *inductive test case* can be viewed which shows a potential execution of the system in which the obligation is unsatisfied. While an unmet test obligation does not necessarily mean that no such test case exists, the inability to find a test case may point to potential issues in the specification.



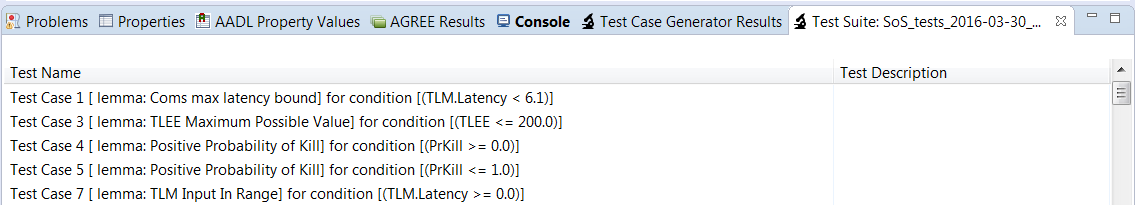
## Navigating to Related Contract Element

Often it is useful to be able to examine the AGREE contract element for which a particular test specification has been generated. This is particularly so when test specifications fail to yield a concrete test case and so are tagged as “Invalid” by the TCG. The context menu provides an option “Go To Guarantee” to easily navigate to the related AGREE contract element:



# Extending and Persisting Test Suites

Our test case generation tool can work with arbitrary test suites. The standard mechanism for constructing test suites involves using the test case generator. After the test case generator runs, it automatically generates a test suite for those test obligations that yield test cases:

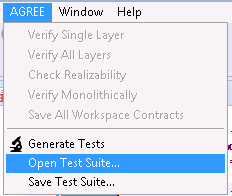


Each test in the suite has a name and an optional description. The name of the tests in the suite constructed from the TCG match their names in the TCG panel. There are several operations that can be performed on tests: they can be renamed, viewed, and simulated. In addition, existing test suites can be extended with additional tests using the simulator.

To view or simulate test cases in the suite, right click the test cases in the suite and select the same from menu options that were available in Chapter 4.

## Persisting / Restoring Test Suites

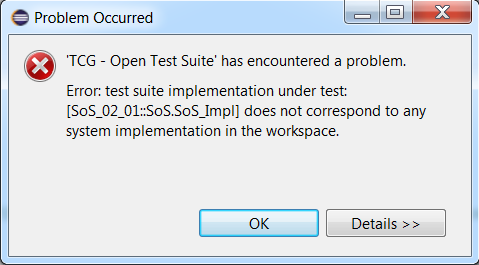
Test suites can be saved in persistent storage and later reloaded using the menu options under the main menu:



The files are stored in XML format which is also shared by the simulator. This facilitates two-way exchange between the test case generator and the simulator, providing an integrated environment for creating test cases from AGREE contracts.

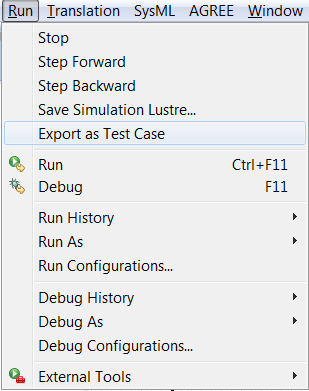
The “Save Test Suite…” menu item persists a test suite into an XML file. It opens a standard Eclipse file save dialog box and allows the user to choose the location to store the generated test suite.

The “Open Test Suite…” menu item opens a previously persisted test suite. Since test suites are generated against a particular system implementation (and usually do not make sense for other models), if the system implementation in question does not exist in the current AADL workspace, the following error is displayed:

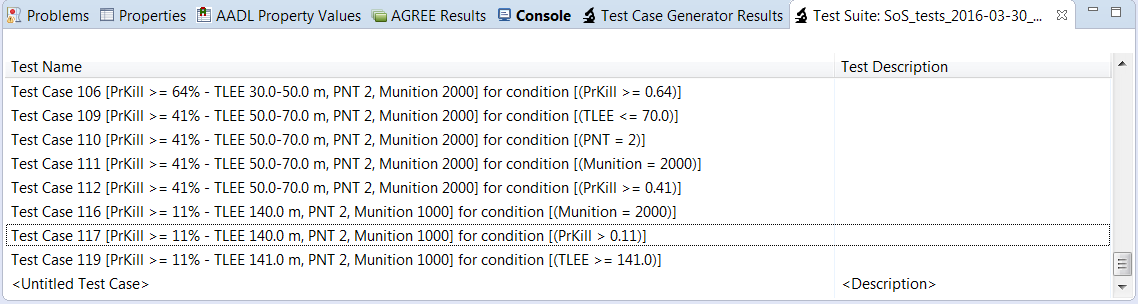


## Extending the Currently Loaded Test Suite with Simulation Results

In order to save simulations as tests in a test suite, one runs the simulator and then chooses the “Export as Test Case” menu item from the “Run” menu:

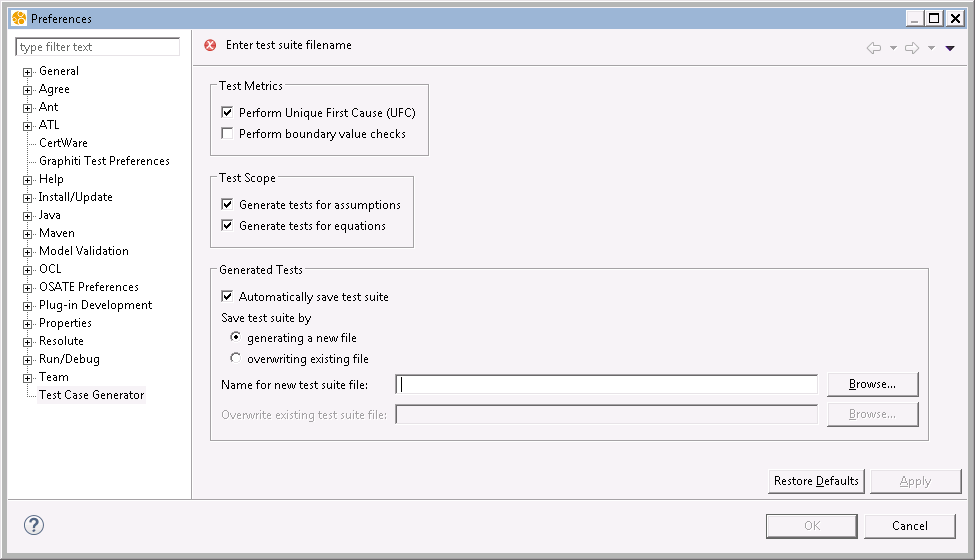


The new test shows up as <Untitled Test Case> at the bottom of the currently loaded test suite, or as the first test in a new test suite is created if no suite is currently loaded:



The new test can be renamed by clicking on the test case and editing the name.

# Test Generator Preferences



The Preferences pane for Test Case Generator can be configured to customize its behavior.

The Test Metrics options determine the test obligations for which test cases are generation. By default test cases are generated to satisfy Unique First Cause coverage. Selecting “Perform Boundary Value Checks” will generate additional obligations by changing numeric quantities by a small margin (1.0). This can be useful for performing boundary value testing to detect off-by-one errors in specifications.

The Test Scope group determines what elements of the AGREE specifications are considered for obligating test generation. By default, only guarantees (and its variants such as lemmas) are considered for coverage with respect to the test metrics. Optionally, assumptions and equations specified in the AGREE contracts may also be included as targets for test coverage.

The Generated Tests groups determine whether test cases are saved in a single file that is rewritten for each generation or whether a new file is generated each time by appending a unique timestamp to a file name prefix. The file name (prefix) and the directory location are specified in this group.