Testing Solutions through SysML / UML

Dave Richards
Artisan Software Tools
Suite 701, Eagle Tower
Montpelier Drive
Cheltenham, UK, GL50 1TA
Dave.Richards@artisansoftwaretools.com

Andrew Stuart
Westinghouse Rail Systems Ltd
PO Box 79
Pew Hill, Langley Park
Chippenham, Wiltshire, SN15 1JD, UK
Andrew.Stuart@wrsl.com

Matthew Hause
Artisan Software Tools
Suite 701, Eagle Tower
Montpelier Drive
Cheltenham, UK, GL50 1TA
Matthew.Hause@artisansoftwaretools.com

Abstract. As systems become increasingly complex and the time to market decreases, systems engineers have to develop novel solutions to testing. The scenario is particularly acute when dealing within the safety critical domain. This paper will seek to highlight how UML and in particular, improvements introduced by SysML can aid the testing process in terms of verification, validation and simulation of software, firmware and mechanical systems. This paper will highlight how UML and SysML constructs can aid testing and is based on many years experience of building and testing systems as well as the experience gained by client companies during consultation. It will highlight on a practical basis how clients have integrated testing into their UML/SysML models to improve their processes and products.

Introduction

The UML is the de facto standard for software development, and allows the user to model their system in terms of structural analysis through class, package, composite structure, object, component, and deployment diagrams. UML also allows the user to model the behavior of a system through state machine, activity, use case, sequence, communication, timing, and interaction overview diagrams. While UML could be used to model systems SysML can more effectively aid the understanding of structure of a system through block definition, package, internal block, instance, parametric, and requirement diagrams. System behavior can further be enlightened investigated and illuminated through state, activity, use case, and sequence diagrams. In particular the use of SysML notation aids the Systems Engineer in construction of systems, system integration and unit tests, as well as providing critical system performance details needed in the handover to software engineers. If we look at each one of these diagrams in turn we can understand what benefits each holds for the Systems Engineer when designing a testable system. See OMG (2007b) for the SysML specification, Hause (2006a) for a summary, and Friedenthal et al (2008), Korff (2008), and Holt (2008) for books on SysML.

Requirements Diagram. The requirements diagram within SysML is a cross cutting construct that allows the system designer to both document system requirements and show the interrelationship that exist between requirements, e.g. refine, verify, satisfy, test case, etc. and other system artifacts (Hause, 2006b).

The Safety Integrity Level (SIL) concept was created in order to quantify the safety requirements of a specific components or set of components in a safety related system (IEC, 2002). Specifically, a SIL is the indicator of the likelihood of meeting required safety features. When we are developing a safety critical system, say to SIL 3, a key requirement is to have full requirements coverage. The requirements diagram can be used not just by the Systems Engineer but by the organization as a whole to model their requirements. The requirements diagram may model requirements from the business case requirements, which could then drive functional requirements that subsequently could be constrained or extended by non-functional requirements. In the latter case we could be specifying an implementation or tool chain. In Figure 1, we see a typical example of a requirements diagram. Note we can link to and exhibit test cases and indicate that a requirement can be verified. In fact the highlighted «test Case» model element “[Package] Maintain Speed - with flows, is in fact a Sequence diagram, from which we can automatically generate code for test sequences.

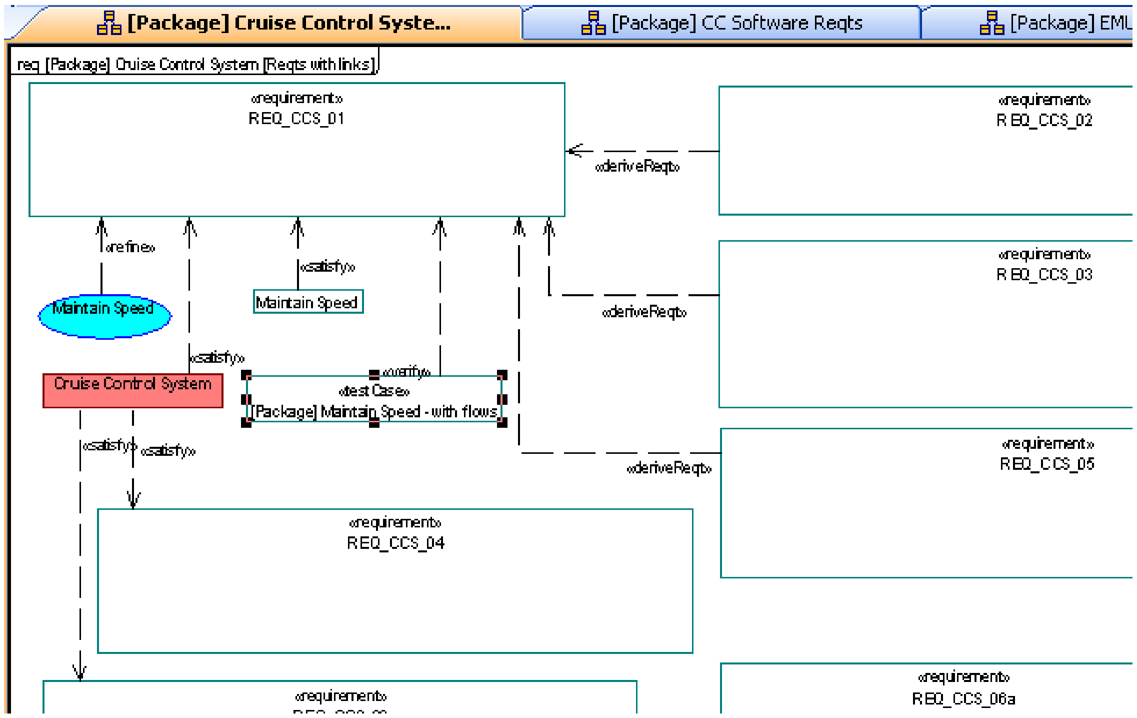

Figure 1. Requirements Diagram with Test Annotations

Activity Diagram. The activity diagram allows low level behavioral modeling of a system. The introduction of <<discrete>>, <<continuous>>, <<nobuffer>>, and <<overwrite>> allows the developer to indicate more precisely the desired behavior of the system. For instance if an object node had <<nobuffer>> applied, and this object node was an interface to a driver which did not have the ability to buffer incoming serial data, a test case could be derived that would exercise the system where the driver had failed to clear the incoming data fast enough. The system engineer would verify whether the system could gracefully recover from such a scenario. Equally we can model an activity edge as either continuous or discrete, and thus derive a more accurate test case in terms of data input / output characteristics. Another scenario where the activity diagram is helpful to system engineers is within destructive wear tests designed to determine when systems fail during extended use. If we apply a probability to an activity edge and then perform an analysis, say, adding a budget stereotype to a physical item associated with the activity, and then perform an analysis of the output, weak points can be identified within the system. You can describe product tests using actions like “open and close the furniture door until the junctions break and count the open action. The test is passed if count is greater than 50000”. Finally, when looking at the low level detail we may be concerned about whether our system properly processes interrupts. By the inclusion of an interruptible region within the activity diagram we can design a test that will ensure that the system recovers from an interrupt gracefully. An example is given in Figure 2.

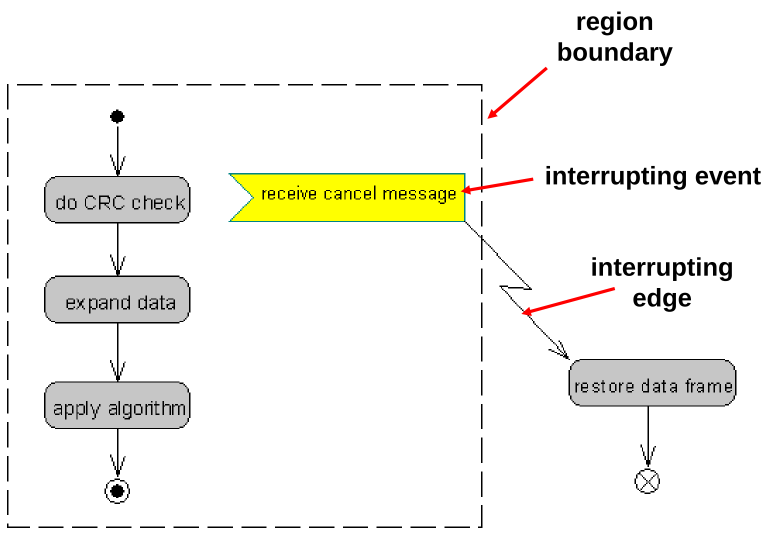

Figure 2. Activity Diagram with Interruptible Region

Block Diagrams. The logical and physical structural characteristics of system are defined using the Block Definition Diagram (BDD) and the Internal Block Diagram (IBD). Both are of great use to the system tester. A BDD for a Cruise Control System is shown in Figure 3.

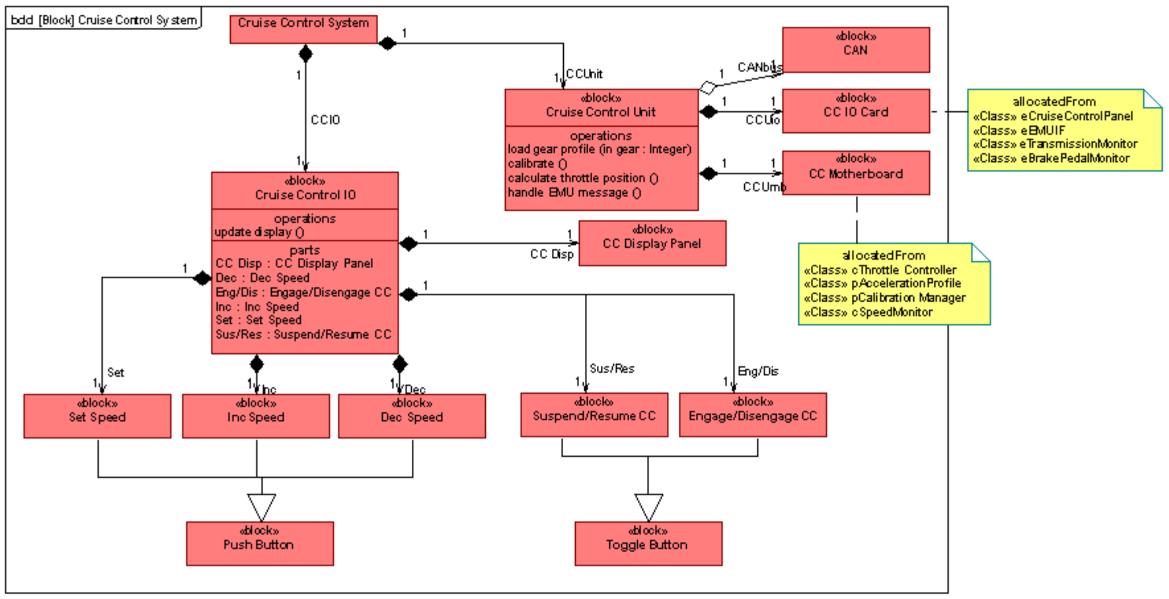

Figure 3. Block Definition Diagram

We can examine the parts of the system that we may wish to test in isolation, and what components interact for an integration test. The block definition diagram lends itself to express real world systems through the rich set of semantics. Looking at the BDD for the Cruise Control unit we see that we may express specialization, composition and associations in an easy to understand format.

While useful in capturing high level detail, the system engineer may wish to provide details about the internal construction e.g. what items are placed on to what buses. The Internal Block Diagram (IBD) can address this issue. An example of this is given in Figure 4. The IBD helps the system engineer to model flow ports, standard ports, standard and required interfaces and the nature of the system interactions. Furthermore, as we add more details to the data model, e.g. in the software design, the bit positions, most significant bit (MSB), least significant bit (LSB), then it is possible to directly generate Interface Control or Interface Definition documents (ICD, IDD) information that can be used as a basis for tests between sub system to sub system or between systems and software.

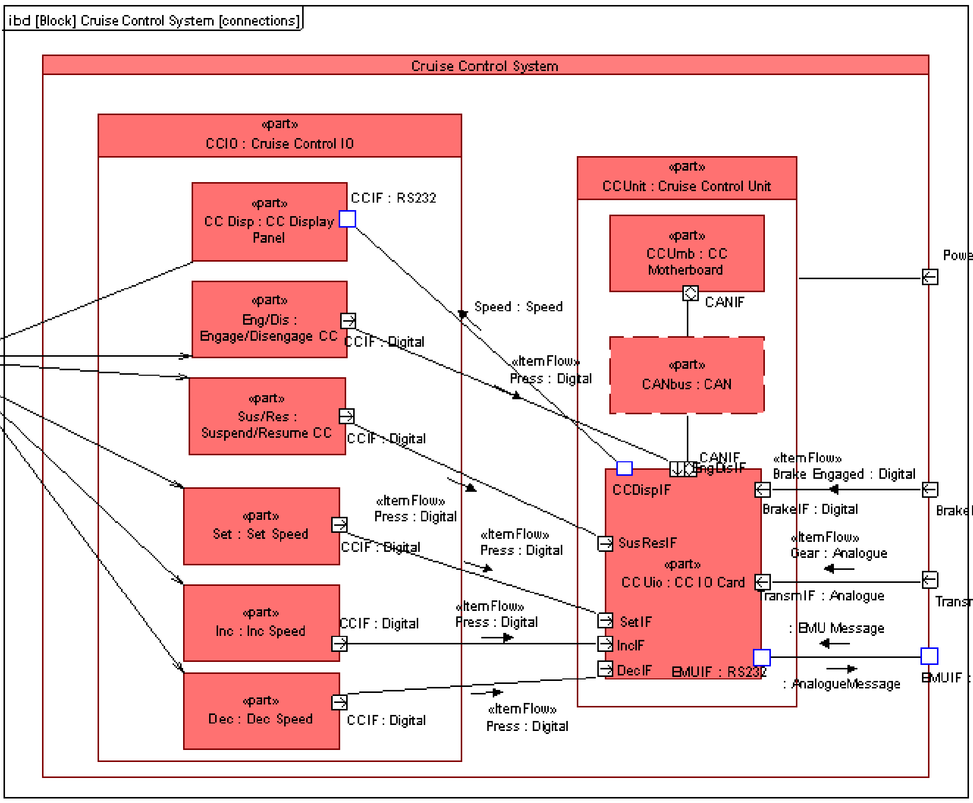
ibd [Block] Cruise Control System [connections]
Cruise Control System
«part» CCIO : Cruise Control IO
«part» CC Disp : CC Display Panel
CCIF : RS232
«part» Eng/Dis : Engage/Disengage CC
CCIF : Digital
«part» Sus/Res : Suspend/Resume CC
CCIF : Digital
«part» Set : Set Speed
CCIF : Digital
«part» Inc : Inc Speed
CCIF : Digital
«part» Dec : Dec Speed
CCIF : Digital
Driver
Speed : Speed
«ItemFlow» Press : Digital
«ItemFlow» Press : Digital
«ItemFlow» Press : Digital
«ItemFlow» Press : Digital
«ItemFlow» Press : Digital
«part» CCUnit : Cruise Control Unit
«part» CCUmb : CC Motherboard
CANIF
«part» CANbus : CAN
CANIF
EngDisIF
CCDispIF
SusResIF
«part» CCUio : CC IO Card
SetIF
IncIF
DecIF
EMUIF : RS232
Power : Electric Power
«ItemFlow» Brake Engaged : Digital
BrakeIF : Digital
BrakeIF : Digital
«ItemFlow» Gear : Analogue
TransmIF : Analogue
TransmIF : Analogue
: EMU Message
: AnalogueMessage
EMUIF : RS232

Figure 4. Internal Block Diagram

Parametric Diagrams. We have so far shown how we can model the interaction of parts that constitute a system in terms of the relationships of physical and logical interfaces. However the parts may also have constraints attached to them. This notation is a specialized IBD and allows the systems engineer to develop rule based specifications in terms of mathematical equations that represent the system behavioral, performance or other constraints. An example is shown in Figure 5. The parametric diagram shows inputs and outputs in terms of value types and the associated mathematical equation. Given this proviso it is possible to then develop black box tests that exercise inputs, in terms of their conceptual representation, say range, domain and value, and outputs that will exercise the equation encapsulating the constraint.

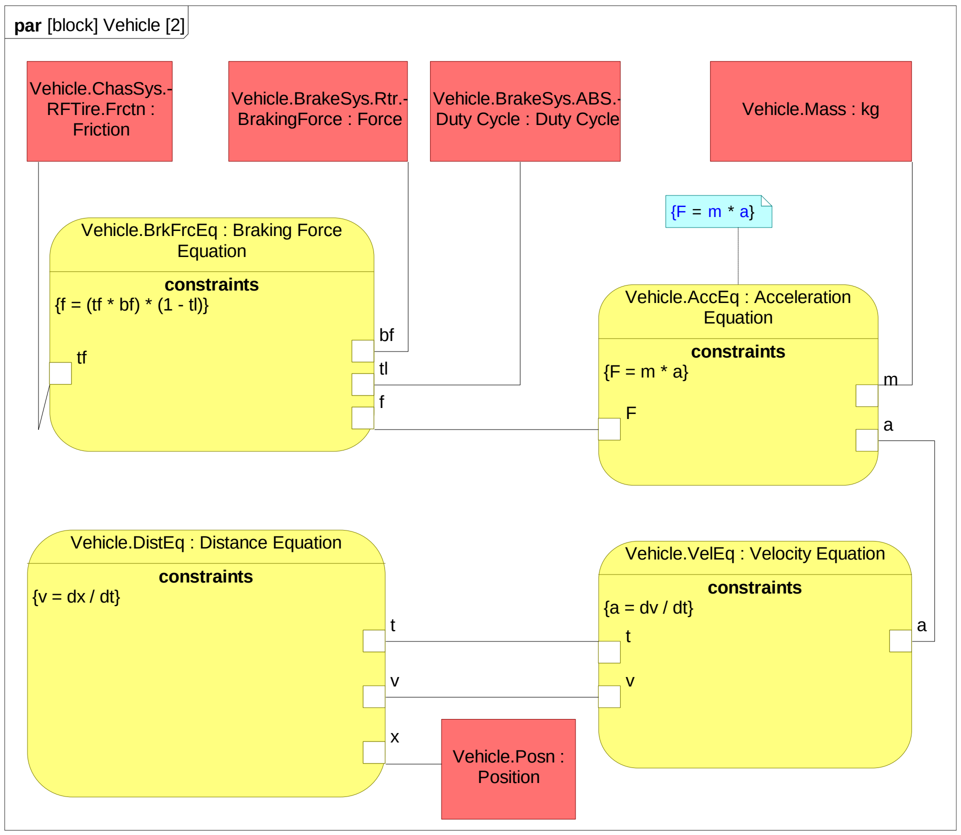
par [block] Vehicle [2]
Vehicle.ChasSys.-RFTire.Frctn : Friction
Vehicle.BrakeSys.Rtr.-BrakingForce : Force
Vehicle.BrakeSys.ABS.-Duty Cycle : Duty Cycle
Vehicle.Mass : kg
{F = m * a}
Vehicle.BrkFrcEq : Braking Force Equation
constraints
{f = (tf * bf) * (1 - tl)}
tf
bf
tl
f
Vehicle.AccEq : Acceleration Equation
constraints
{F = m * a}
F
m
a
Vehicle.DistEq : Distance Equation
constraints
{v = dx / dt}
t
v
x
Vehicle.VelEq : Velocity Equation
constraints
{a = dv / dt}
t
v
a
Vehicle.Posn : Position

Figure 5. Parametric Block Diagram

Allocation. While the BDD and IBD provide a visual representation of the system under construction they are also useful for understanding the allocation of parts within the system. Understanding allocation helps the tester identify the inter relationships between functionality, requirements and other blocks as different perspectives of the same system. An example is given in Figure 6. The SetSpeed block is allocated to the SetSpeed operation and SetSpeed part has been allocated from CruiseControlIO.

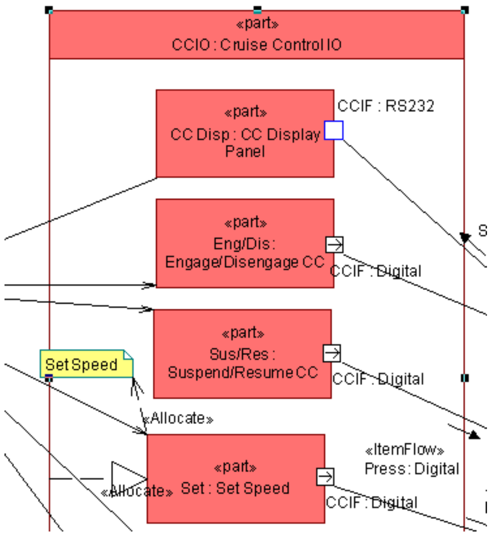

Figure 6. Allocation of concerns.

Summary. SysML introduces a number of new diagrams, constructs and semantics that help the system engineer in designing a testable system from the outset. When we design a system for test there are numerous factors that have to be taken into account, from clear dissemination of concerns, to specifying clear and well understood interfaces. Consequently ICD documentation can automatically generate a basis for system integration tests. If we choose from the outset to specify a system using SysML we can naturally then express software in the form of UML. From here test cases, code, and the accompanying documentation may be generated. Taking this approach reveals a flexible system which also reduces verification and validation costs. We have shown that in each stage of systems engineering we may use the SysML to design a more testable system and that at every stage from system conception to ironing out the detail at the low level interface SysML is the key element in the systems engineering toolkit. Examples of how this has been achieved in industry will now be highlighted.

Case Studies

Rail Signaling System Test. Within the European rail domain, software is developed according to EN 50126, Railway Applications: The Specification and Demonstration of Dependability - Reliability, Availability, Maintainability and Safety (RAMS). This is used in conjunction with EN 50129 Railway Applications: Safety related electronic systems for signaling, and EN 50128 Railway Applications: Software for Railway Control and Protection Systems. These norms have been created by CENELEC (Comité Européen de Normalisation Électrotechnique), and therefore are often referred to as "CENELEC-norms" in the rail domain. In this example, we look at the deployment of a Radio Block Control (RBC) for the European Rail Traffic Management System (ERTMS) Malaga line. The RBC is responsible for continuous speed supervision and movement authority of the train. The train uses Eurobalises to determine the Train Location and sends it to the RBC. Eurobalises are track mounted devices that operate on transponder technology. The balise transmits information to the train, such as: location of the balise; the geometry of the line, such as curves, gradients and speed limits; and the position of any signals. Balises are typically deployed in pairs so that the train can determine its direction of travel A->B from direction B->A. The Interlock authorizes and gives proceed authorization to the RBC to allow train movement. The RBC is categorized as a SIL 4 product. This scenario is shown in Figure 7. One aspect of system validation requires the analysis of all possible execution sequences in order to gain full test coverage.

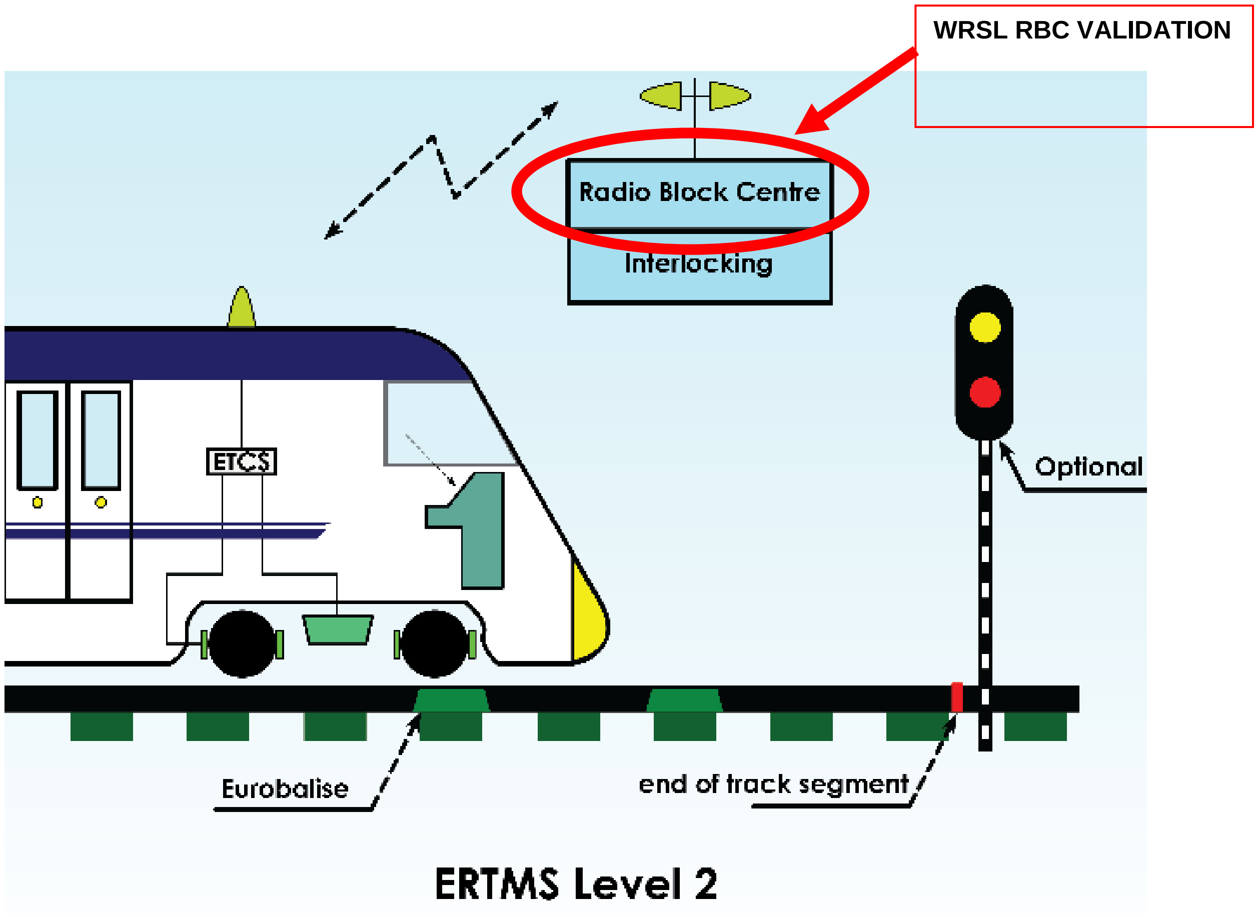

Figure 7 RBC Validation Scenario

Traditional approach to performing RBC Validation. In the approach illustrated in Figure 8, the System Requirements Specification (SRS) was modeled in a UML Use Case model within Artisan Studio. Design and analysis work continued within Studio and were expressed in terms of UML State and Sequence Diagrams. As Studio provides an OLE interface, an in-house tool was developed that allowed the construction of flow charts from the State and Sequence Diagrams that captured all possible execution paths. The information was then exported in the form of a spreadsheet. The validation team then generated test case SDs that exercise the system. The validation team identified which paths are valid and generated test cases for these, which paths are impossible to run, and which paths are incompatible. Only paths that are valid had test case SDs generated. With the test case SD constructed within Studio, it was possible to develop a custom code generator that interlinked into the Studio model. In this instance Python scripting language was chosen to exercise the system under test (SUT). The Python script exercised the system under test on both a simulated PC platform and/or on a physical test rig. Finally test report documentation was constructed and presented to the Railway Authority as certification evidence. While this overall approach was more efficient - all valid paths of the product are exercised, it did have the drawback of not being very flexible if the original UML test model elements such as state diagrams and sequence diagrams changed.

There is a solution to this issue, namely Automated Validation.

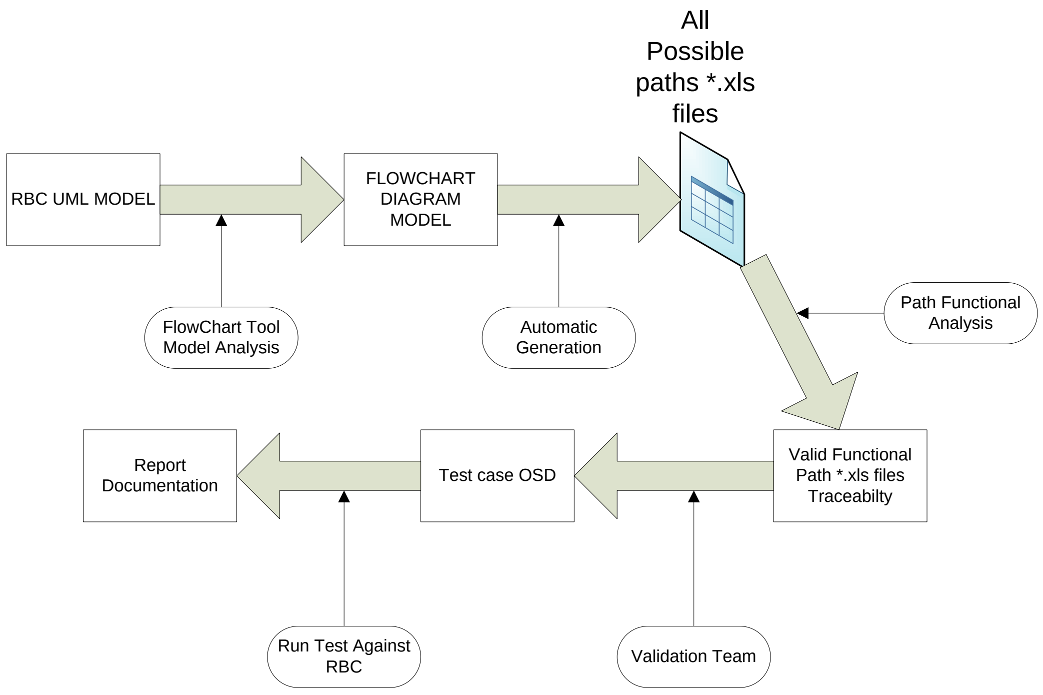

Figure 8 Traditional RBC Validation

RBC Automated Validation. The automated validation approach is outlined within Figure 9.

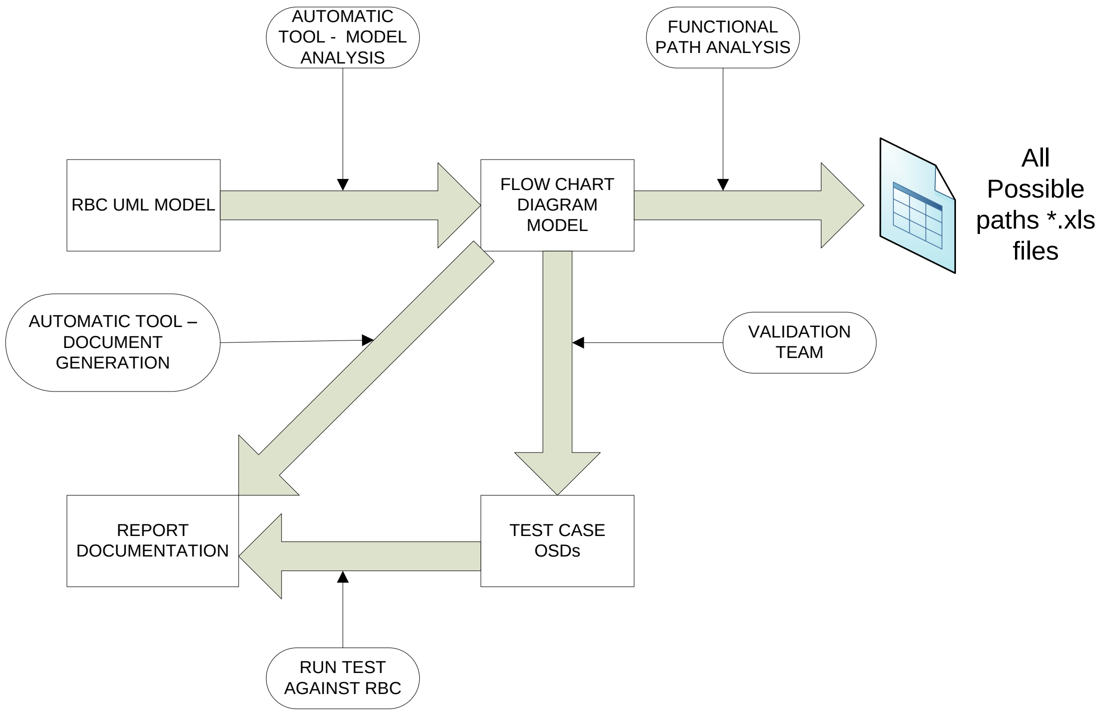

Figure 9 Automated RBC Validation

The main objectives of the automated validation process are:

1) Eliminate the amount of manual work.
   The excel files are created automatically. This information was included as evidence.
2) Reduce human factor influence (human errors) in validation.
   Originally these files were hand-coded.
3) Decrease the number of files used in the validation process.
4) Enforce design standards.

The tool integration for this process is shown in Figure 10. The Visual BASIC DLL reader imports information from the Artisan Studio model via the OLE interface, namely SDs and State Diagrams. Information contained within SDs includes text, sequence order and links in steps. For State Diagrams, levels in diagrams, State Transitions, input /output functions and text are imported. The Visual .NET control application manages the imported information and creates flowcharts and state diagram representations. The control application tool analyzes this information to provide all possible execution paths, but also allows the user to modify the paths created by allowing the addition of steps, states, transactions etc. Incompatible paths can be detected as guided by the user. As the lifecycle of the project progresses, different versions of the diagrams can be compared for changes. Finally the application automatically produces validation documentation. Thus at this stage our workflow amounts to the validator analyzing execution paths in a graphical manner. This process produces a 'possible paths.xls' spreadsheet and documentation templates that include the steps executed in the test case. Finally the design standard for model construction is enforced as the control application tool compares the SD and State Diagram construction against the accepted quality standard, e.g. conditional expressions within the SD defined since UML 2.0 using fragments.

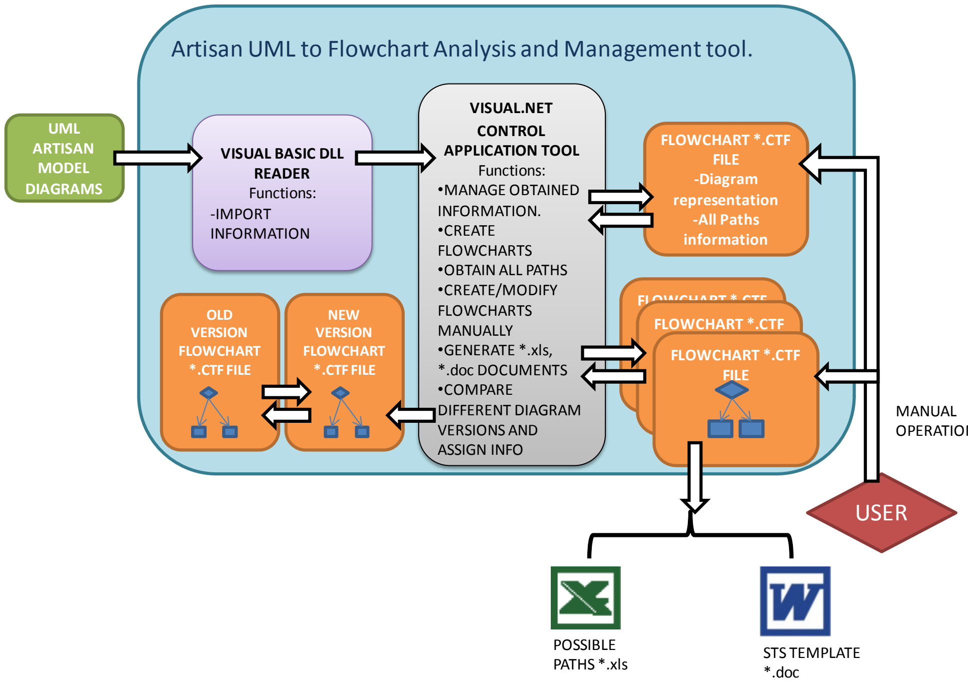
Artisan UML to Flowchart Analysis and Management tool.
UML ARTISAN MODEL DIAGRAMS
VISUAL BASIC DLL READER
Functions:
-IMPORT INFORMATION
VISUAL.NET CONTROL APPLICATION TOOL
Functions:
•MANAGE OBTAINED INFORMATION.
•CREATE FLOWCHARTS
•OBTAIN ALL PATHS
•CREATE/MODIFY FLOWCHARTS MANUALLY
•GENERATE *.xls, *.doc DOCUMENTS
•COMPARE DIFFERENT DIAGRAM VERSIONS AND ASSIGN INFO
FLOWCHART *.CTF FILE
-Diagram representation
-All Paths information
OLD VERSION FLOWCHART *.CTF FILE
NEW VERSION FLOWCHART *.CTF FILE
FLOWCHART *.CTF
FLOWCHART *.CTF FILE
MANUAL OPERATIONS
USER
POSSIBLE PATHS *.xls
STS TEMPLATE *.doc

Figure 10. RBC Tool Integration

Most important of all, the adoption of this process has led to a decrease of about 75% in the cost of validation of this SIL 4 product. This demonstrates a significant return on investment (ROI) for this approach to model-based testing.

Automotive Model Based System Test. Often within the automotive industry a standard application is deployed on a different number of target platforms. Traditional systems would require extensive testing for each of these platforms, often carried out by hand. This is time consuming and subject to significant delays and re-analysis should alterations be introduced when testing has commenced. There is a solution to this problem though: Model Driven Development with automatic test script production. So the same principles for model based software development are used for test script generation. An overview of the platform independent modeling system is shown in Figure 11.

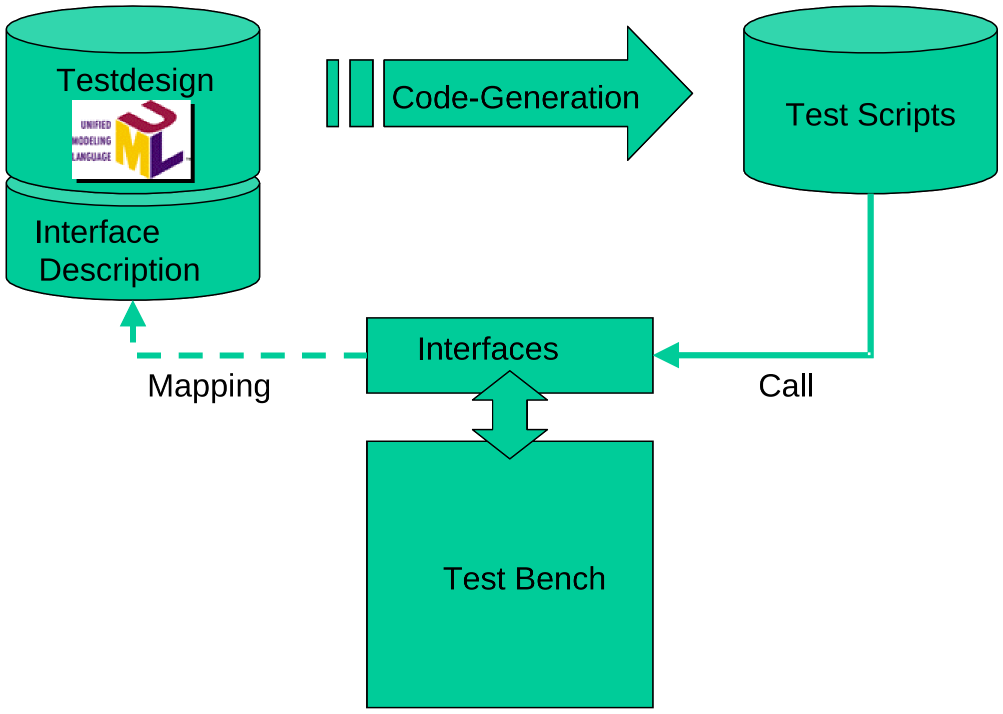

Figure 11 Automotive Platform Independent System Test.

The system produces, administers and executes the test scripts. The system as a whole exercises Hardware In the Loop (HIL) testing. Testing breaks down into 3 stages.

1) Model the test case, i.e. create the test case, define the parameters for the test, and model the test sequence.
    a. Test Cases are derived from Use Cases (usually black box, but not always)
    b. White box Test Cases can be derived (deduced) from Activity Diagrams
    c. Test Scripts can be derived from both Activity and (more often) Sequence Diagrams
2) Automatically generate the test case, through a code generator based on model content.
3) Execute the test on the test rig, capture results and then automatically produce test results documentation.

Figure 11 shows a high level view of the components. The test bench shown in Figure 12 provides a configurable remote test model of the car. The interface to the RTOS and application under test is through predefined standard test interfaces.

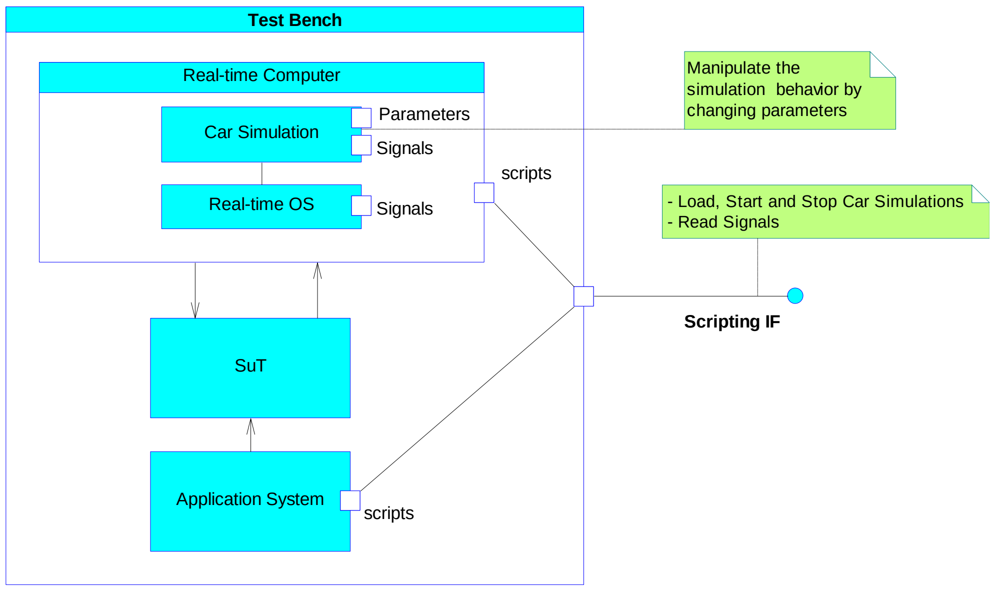

Figure 12 Testbench Overview

The test sequences are defined in UML Sequence Diagrams and are platform independent this is possible due to the fact that the test library contains the operations of the actual test equipments used and their abstracted functionality modeled as interfaces. These interfaces are used in the test sequences, thus allowing the mapping of the test to the equipment to be used separately. The test bench computer interacts with the test bench, as per Figure 13. The shown dependencies depict the fact that the mapping between variables used in the test descriptions and the parameters and signals of the simulation are pragmatically realized using name matching. The test function library holds drivers that communicate with the relevant platform.

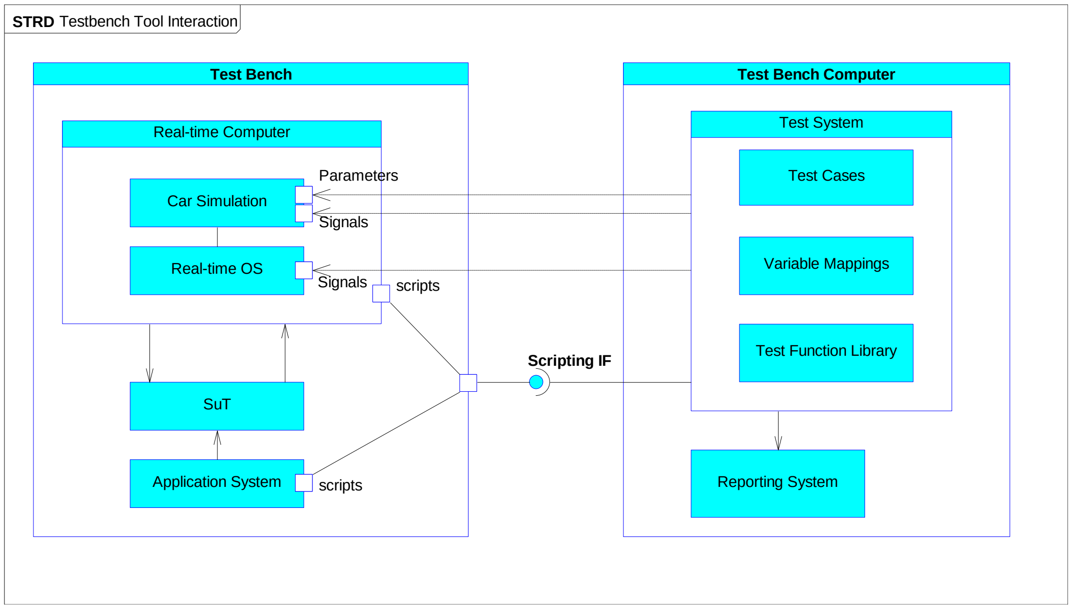

Figure 13 Testbench Tool Interaction

While we have highlighted the functional components, we need to understand how data flows through our system. This will also serve to highlight tool integration. This is highlighted in Figure 14. The central component within this test system is the Artisan Studio repository. Studio allows an XML export of the test case design. This has two purposes. Firstly, XML serves as an input to a code generator, python in this case. It also provides a base for the documentation of the test cases. Using a model driven approach simplifies the test case generation and allows the validator to concentrate on the goal of the test case as opposed to being caught up with the implementation details. As an alternative, the production of XML can be omitted, resulting in a better performance when generating the test scripts directly from the model. This was done in a second phase of the project

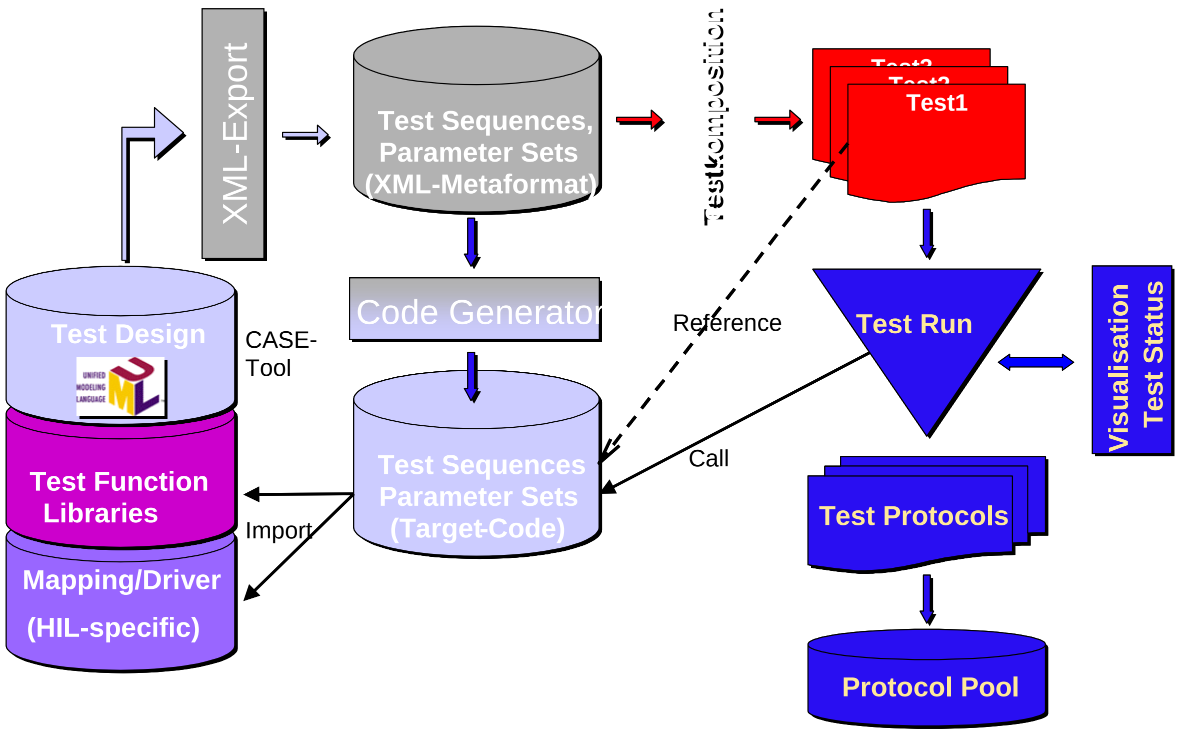

Figure 14. Automotive Tool Integration and Workflow

As a summary, figure 15 shows how the test scripts formerly produced manually by the testers are abstracted in the model-based test design. The scripts contain all levels of information, from the test strategy to the very specific access to different elements of the test bench. In the models, there are explicit perspectives for these levels. Use Cases manage the test cases and allow the combination of test parameters and test sequences, both of which are modeled using UML interaction modeling. The test infrastructure is described using the UML class model, which contains the ability to abstract each equipment type to the relevant service it provides for a test. The automated test script generation compiles the tests from the use case, i.e. test management level, the interaction model, i.e. the test sequences and parameters and the description of the test infrastructure into many low level test scripts, which can be run in the test automation infrastructure automatically.

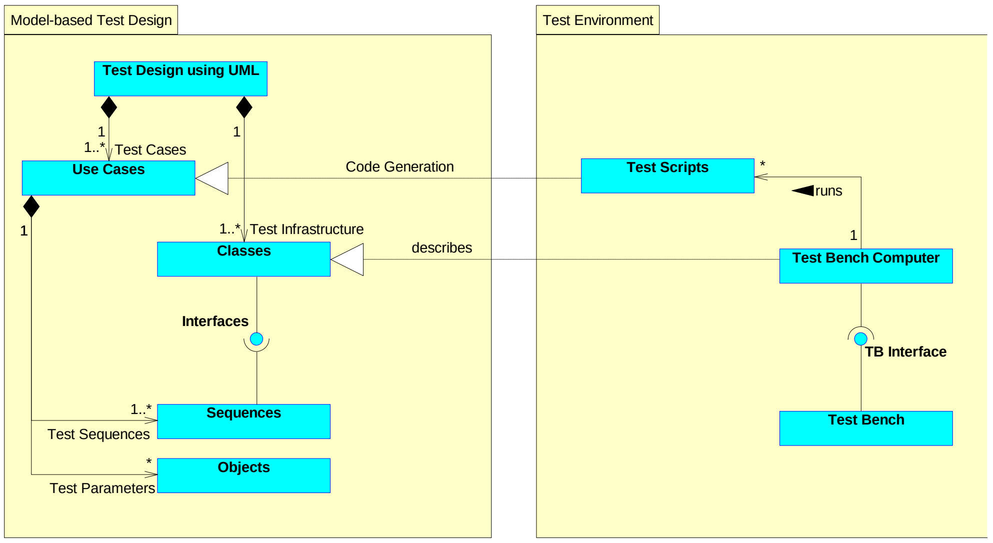

Figure 15. Description of Automotive Model-based Testing

Overall using a model driven test approach decreases the amount of human interaction, and probability of human error, while still allowing the validator to focus on the task at hand. This leads to more efficient and flexible system that can be reused as new products and interfaces are introduced.

Remark on standards for Model-based Testing

Since 2005, there is a OMG standard profile available for modeling test, called the UML 2 Testing Profile (U2TP). In general, both case studies used the same or a similar approach like the one supported by U2TP, but without explicitly using this profile. Since SysML and also the U2TP are based both on UML 2.0, systems modeling and test modeling using U2TP can be used combined in one model. The SyML stereotype «testCase» is compatibly defined to comply with the same concept in U2TP.

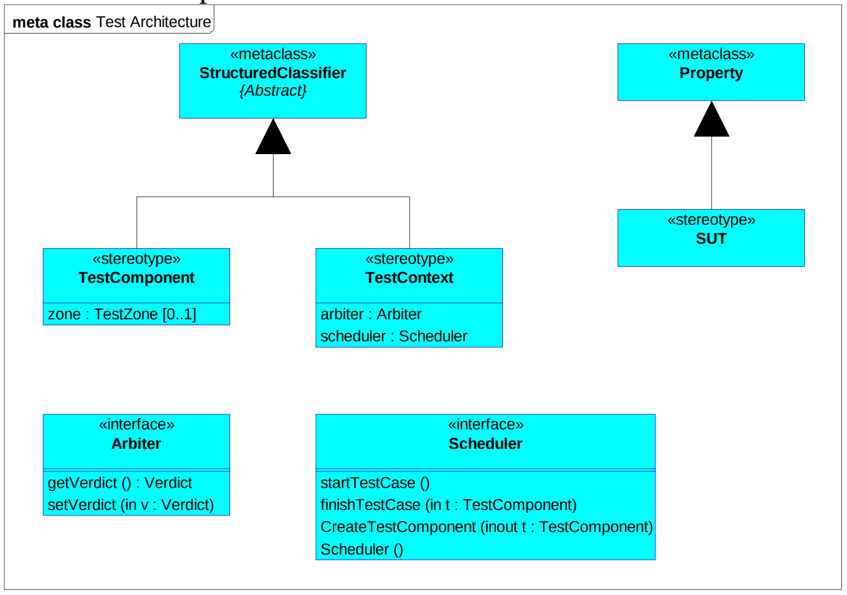

Figure 16: U2TP Test Architecture

A test starts with setting up a test context. This contains an arbiter and a scheduler, which can be derived from the library elements shown in figure 16. The test case itself is defined in the UML behavioral model, so it could be a behavior or an operation, like stated in figure 17, showing also the four different test results. A test can be passed, failed, there could be a test error or the result is unclear, so there might be an error in the test concept itself.

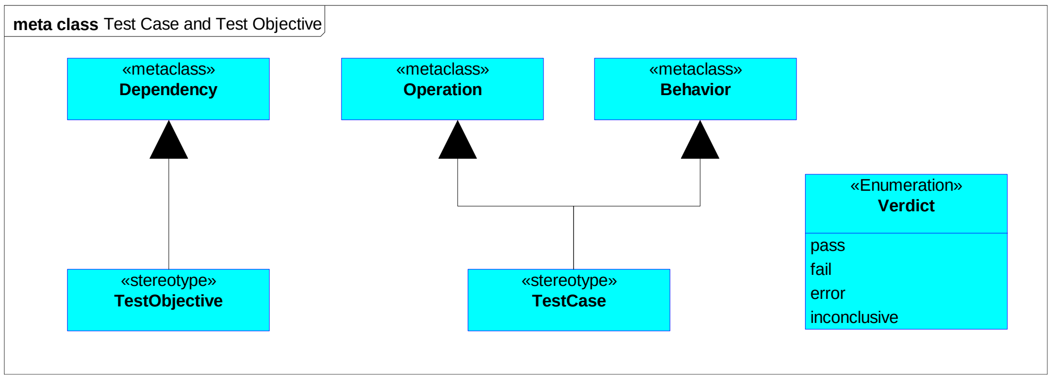

Figure 17: Test Case and Test Objective

The test sequence itself is mainly using a sequence diagram, which in general has three areas: Test Setup, test run and gain test results. Since sequence diagrams can be structured now, there are all means available to model even very complicated tests, but leverage the complexity by e.g. referencing subsequences.

The level of automating test runs is not covered by the U2TP, because it highly depends on the tool chain used. Examples for successful application of this model-based testing approach in real projects are shown in the above described case studies. However, it is always good to know that there is a theoretical basis available for the modeling ideas pragmatically used.

Conclusion. Test ‘Plans’ can and should be modeled. The test plans should define the nature and purpose of the test suite (a collection of test cases). For example Basis Path or Coverage Testing Plans (usually white box) should specify (or model) the arcs and nodes (paths and modules) traversed and exercised by the suite. Parametrics can also be specified (and modeled) in terms of input-parameter-to-output-results tuples and automatically verified within the model itself (or an external script). The logical progression of:

Test Plan (and models) to,
Test Suite(s) (and models) to,
Test Case(s) (and models) to,
Test Scripts (and models) is in and of itself a “successive refinement” and decomposition process that lends itself even more naturally to symbolic modeling than do the more typical analysis and design activities. That is to say it is in actuality easier & more natural to model and build test plans and suites symbolically than systems and software engineering analysis and design. It also results in demonstrable ROI.

References

Czarnecki, Eisenecker, Generative Programming: Methods, Tools and Applications, Published by Addison Wesley.

Friedenthal, S., Moore, A., Steiner, R. Practical Guide to SysML: The Systems Modeling Language, Morgan Kaufman September 2008

Hamil, Unit Test Frameworks, Published by O'Reilly.
Hause, M.C., 2006a, The Systems Modeling Language - SysML, Sept 2006, INCOSE EuSEC Symposium, Edinburgh, 2006 Proceedings.
Hause, M. C., 2006b, Cross-Cutting Concerns and Ergonomic Profiling Using UML/SysML, INCOSE International Symposium Orlando, Florida, Proceedings.
Holt, J., Simon Perry, S., SysML for Systems Engineering, IET Publications, 2008
IEC, Functional safety and IEC 61508 A basic guide, November 2002, The International Electrotechnical Commission (IEC) available online from http://www.iee.org/oncomms/pn/functionalsafety/HLD.pdf
Korff, A., Modellierung von eingebetteten Systemen mit UML und SysML, von Spektrum Akademischer Verlag Taschenbuch - 13. June 2008
Object Management Group (OMG), 2005a, UML Testing Profile for UML 2.0, v1.0, formal/05-07-07 (full specification)
Object Management Group (OMG), 2005, Military Architecture Framework Request for Information, Available from www.omg.org. [Accessed April, 2005]
Object Management Group (OMG), 2007a. Unified Modeling Language: Superstructure version 2.1.1 with change bars ptc/2007-02-03. [online] Available from: http://www.omg.org [Accessed September 2007].
OMG Systems Modeling Language (OMG SysML™), V1.0, 2007b, OMG Document Number: formal/2007-09-01, URL: http://www.omg.org/spec/SysML/1.0/PDF, Accessed November, 2007
Steven, Jackson, Brook and Arnold, Systems Engineering: Coping with Complexity Published by Prentice Hall.

Biography

David Richards, Application Engineer –Artisan Software Tools

David has worked in the Safety Critical domain for 15 years. Starting with financial systems, and moving to RTOS Operating System development, and successfully completed numerous consultancy projects in both civil and military aviation as well as rail signaling projects. His current role includes the specification of tool integration projects as well as sales presentations and training courses.

Andrew Stuart, R&D Systems Engineer – Westinghouse Rail Systems Ltd

Andrew Stuart has worked as a systems and software engineer on many different safety critical domain projects for most of his career including rail and avionics.

Matthew Hause, Chief Consultant at Artisan Software Tools

Matthew Hause is Artisan's Chief Consulting Engineer, is a member of the OMG SysML specification team, and the co-chair of the UPDM group. He has been developing real-time systems for over 30 years. He started out working in the Power Systems Industry, and has been involved in Process Control, Communications, SCADA, Distributed Control, military systems and many other areas of real-time systems. His roles have varied from project manager to developer. His role at Artisan includes mentoring, sales presentations, standards development and training courses. He has written a series of white papers on project management, Systems Engineering, architectural modeling and systems development with UML, SysML and Architectural Frameworks. He has been a regular presenter at INCOSE, the IEEE, BCS, the IET and other conferences. Matthew studied Electrical Engineering at the University of New Mexico and Computer Science at the University of Houston, Texas.