FP7 Project 2007- Grant agreement n°: 218575

Project Acronym: **INESS**

Project Title: **INtegrated European Signalling System**

Instrument: Large-scale integrating project
Thematic Priority: Transport

WS D – Generic Requirements

Deliverable D4.2.1 – Documented methods for expressing test cases in UML

Due date of deliverable	March 2010
Actual submission date	July 2010

Deliverable ID:	**D4.2.1**
Deliverable Title:	Documented methods for expressing test cases in UML
WP related:	D4
Responsible partner:	UIC
Task/Deliverable leader Name:	UIC
Contributors:	C. de Courcey-Bayley, K. Agrou

Start date of the project: 01-10-2008 Duration: 36 Months
George Barbu De Cicco
Project coordinator organisation: UIC

Revision: WS Final Dissemination Level[1]: CO

[1] PU: Public, PP: Restricted to other programme participants (including the Commission Services), RE: Restricted to a group specified by the consortium (including the Commission Services), CO: Confidential, only for members of the consortium (including the Commission Services).

Document Information

Document type:	Report
Document Name:	INESS_WS D_Deliverable D4.2.1_Documented methods for expressing test cases in UML
Revision:	WS Final
Revision Date:	26-10-2010
Author:	C. de Courcey-Bayley, K.Agrou / UIC
Dissemination level:	**CO**

Approvals

	Name	Company	Date	Visa
WP leader	K. Agrou	UIC	31-03-2010	validated
WS Leader	W. v. Spronsen	ProRail	22-10-2010	validated
Project Manager	E. Buseyne	UIC	26-10-2010	validated
Steering Board				

Document history

Revision	Date	Modification	Author
1st	23/09/2010	TAB comments processing	K. Agrou
WS Final	26-10-2010	Format and quality checking	Richard Vaux, PMO

TABLE OF CONTENTS

GLOSSARY

HAL:	Hardware Abstraction Layer: the internal part of the interlocking where the physical elements are mapped
OMG:	The Object Management Group http://www.omg.org/
SUT:	System Under Test
xUML:	eXecutable Unified Modelling Language

Section 1 – EXECUTIVE SUMMARY

INESS aims at delivering the functional requirements of an ERTMS-compliant interlocking system. Once the functional requirements are captured, the resulting common core of functionalities must be tested in order to ensure the correct interpretation of the functionalities expressed. This document presents a methodology for the expression of test cases in a formal way, and is part of the research dimension of INESS. Since functional requirements are expressed in SELRED and afterwards modeled in xUML, it was logical to search for a methodology that remains implementation-independent and compatible with UML: this is the purpose of the model-driven testing methodology presented here.

Section 2 – INTRODUCTION

Even the most cursory glance at the V-Model makes it clear that model-based development of safety-critical systems cannot be completed without being able to express and perform tests. However, the V-Model only foresees testing being undertaken *after* the construction phase, at the level of system testing. Given that we know, testing complex systems usually comprises a large part of the overall effort and cost of development, it seems sensible to try and develop a method to enable the testing process to be initiated much earlier in the project. Thus it was decided in the INESS project to capture the requirements in a manner that allowed them to be tested *as requirements* rather than only once the specified system had been implemented. This would mean that the quality of the requirements could be raised and that some input could be made to the later testing at the system level. This approach is illustrated in figure 1 below

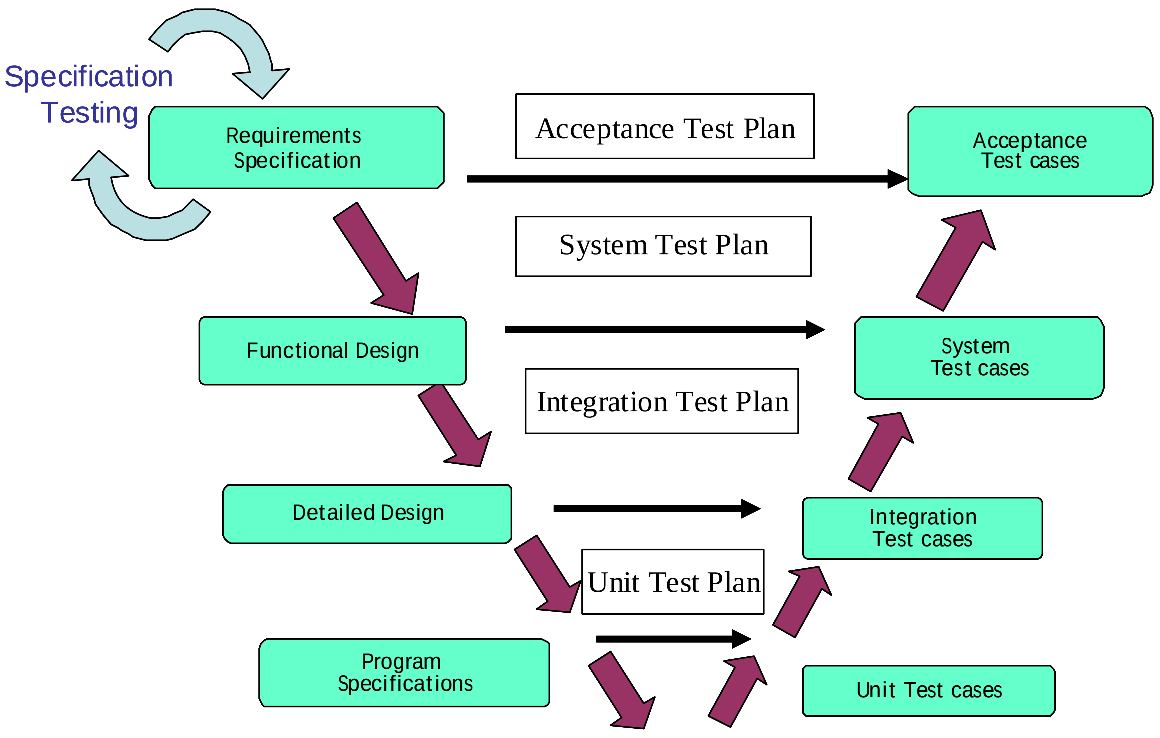

Figure 1

As a consequence of this desire to raise the verified quality of the requirements, it was decided not just to catalogue the project's various railways' functional requirements, but also to model these requirements formally with the view of being able to then validate and verify them. The method for the translation of structured English-language requirements into UML was taken over from the Euro-Interlocking project, as much progress had been made in the forerunner project in the field of requirements modelling and this method was documented in two previous INESS deliverables; D.D.1.2 the Requirements expression document and D.D.1.5 the report on the translation of requirements from text to UML.

Although the Euro-Interlocking project produced a catalogue of textual functional tests derived from the functional requirements, no work was ever undertaken to express these tests formally in UML; thus in order to complete task D.4.3.1 (creating a core set of validated functional requirements), a method for validating the requirements should be produced by the INESS project. This paper is a draft for a testing method analogous to that already defined for the translation of the written requirements into UML and its outcome should be a process that will enable generic test cases to be modelled in UML from which specific tests can then be configured and executed.

During the Euro-Interlocking project, the process of modelling the requirements in itself often provided useful input to clarify the given requirements. However, any improvements that resulted from the modelling process cannot claim to have been comprehensive; as at no point was testing undertaken in a structured manner. However, to now create a formal process to define generic use cases that can be made specific for a given layout at run time is really nothing else than a natural conclusion of the whole modelling process.

The idea presented here is not merely to create more modelled artefacts, but to embed the whole specification testing environment into the model as a whole. Thus, given judicious use of both the UML artefacts that anyway exist in order to represent the requirements and of the data configuration by which the layout of a particular station area is expressed, it should be able to extend the model with its own suite of generic test cases that can be populated similarly to how a specific railway layout is simulated from the UML model.

Before actually doing this work, it has first to be defined both what would be useful and also what is actually feasible. There is a broad spectrum of possible results which range from creating a testing “method” that identifies ‘things of interest’ to test all the way to actually performing every single conceivable test for a given layout and reporting on whether the test was successful or not. At the outset it should be stated, that the latter is clearly not the scope of this deliverable, as it is in fact nothing else than full-blown model checking and this is the task of D4.1. On the other hand, if the method was merely to propose a UML-compatible syntax by which the conditions of the various elements involved in a test case could be expressed, this would do little to raise the quality of the modelled requirements themselves.

Thus it is the goal of this paper to propose what such a method needs to do and also to make concrete suggestions as to how the proposed method will fulfil its task. Central to this is the aspiration to notate test cases in such a manner that they will be re-usable and also to propose how they could be used in the context of actual structured testing of the simulation. It should not be assumed that the final deliverable of D.4.2.1 is yet clear and this paper should only be seen as a milestone on the way to achieving it and in so doing it should both taken especial consideration of what is both useful and what is feasible.

Section 3 – TESTING METHOD

3.1 Preliminary remarks

Originally UML provided no means to precisely describe a test model and a testing profile was only subsequently was added to UML 2.0. This ‘UML 2.0 Testing Profile’ became an official OMG standard in 2004. The testing profile bridges the gap between the engineers who specify a system and those who test it by providing a means within UML for creating both the model and the test specification. This allows the UML requirements artefacts to be reused for testing and also enables test development to commence at an early phase of system development. Both of these are what INESS is seeking to achieve. The benefit of being able to do this is that as the specification can be tested, it is hoped that the quality of the model (and textual requirements on which it is based) can be improved.

The modelling work of the project and of the Euro-Interlocking project that proceeded it was carried out using the Artisan Studio Case Tool. An advantage of using this tool is that – with certain extensions and manipulations – it can also support the OMG UML Testing Profile, which means that there is no need to transform or manually exchange information between different domain models to enable testing, i.e. in as far as elements modelled for the requirements are useful and relevant for testing, they will be available for use in the defined test cases. This has both the practical advantage of only requiring certain information to be captured once and enabling it to be reused as opposed to rewritten; this naturally reduces the chance of transcription errors and the usual problems associated with having information captured redundantly.

However, before looking at the testing profile in detail, it is worth seeing how it fits into the whole modelling context. The existing process for the creation of an executable specification of a specific track layout can be graphically presented as follows:

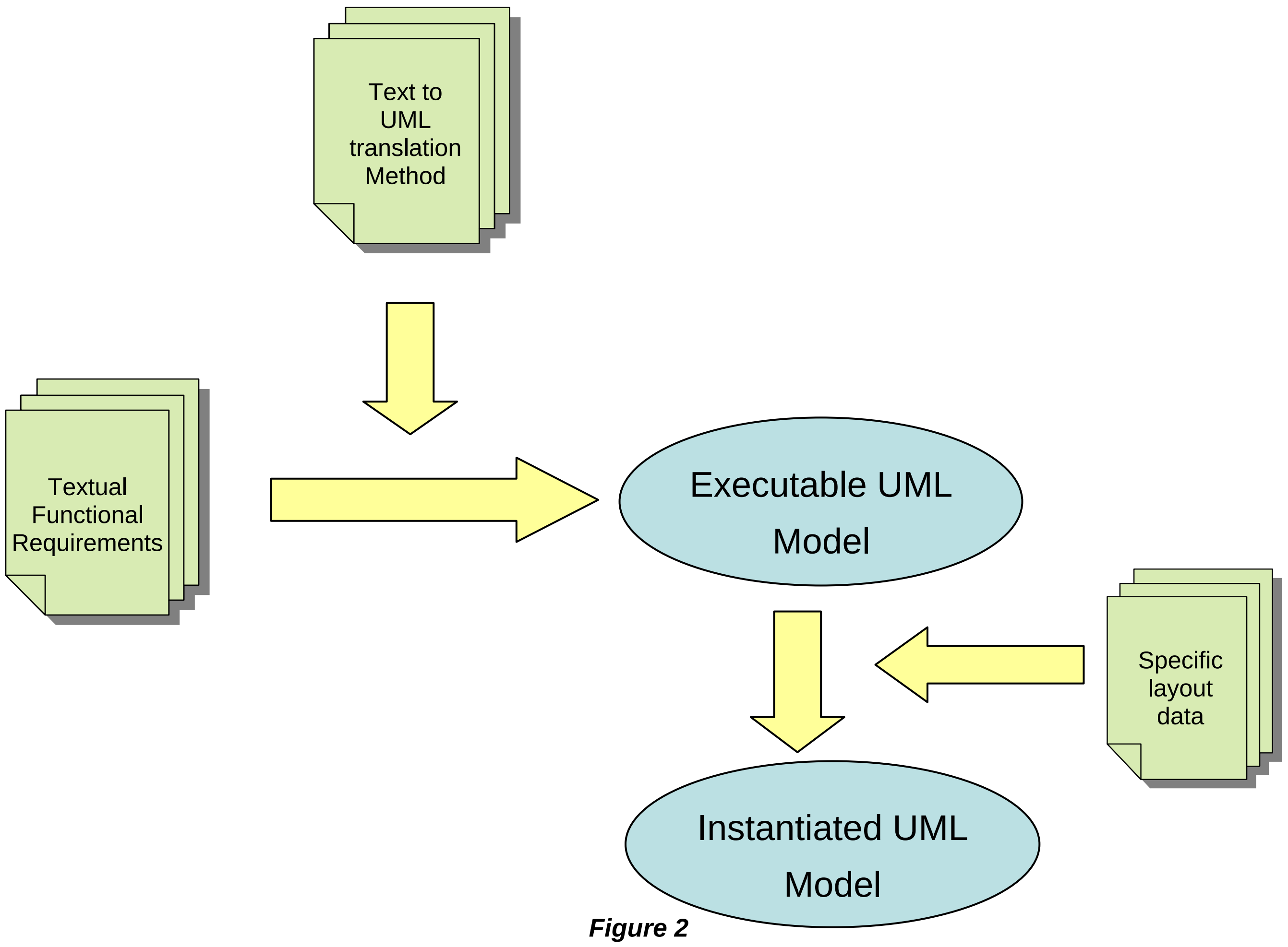

Figure 2

The method for achieving these steps has been laid out in previous INESS documents and a substantial amount of work based on this method was implemented during the Euro-Interlocking project. Fundamentally speaking, the goal of this document is to establish the procedure by which a similar method can be created for the test procedure.

The method for defining the tests in UML will be similar to that of modelling the requirements in the first place. For just as the requirements can only be seen in action when instantiated together with a specific layout configuration, the modelled test cases shall also be generic and only populated with specific test data at run time. A pre-condition for this to happen is that generic test cases can be derived from the requirements which can be then used as the template for the creation of the modelled test cases.

3.2 UML Testing Profile

As stated above The OMG UML testing profile was designed to support implementation-neutral system testing. It is based on the existing UML meta-model and it enables the specification of tests for both structural (static) and behavioural (dynamic) aspects of UML models.

The profile itself foresees the use of a number of concepts in defining a test environment, the three most useful of which in the context of the INESS project can be summarised as follows:

- the architecture of the test
- the actual behaviour desired in the test
- the data to be used in the test.

Together these concepts allow the specification and visualisation of a test system that can be used on a system and (crucially in this case) which is in itself analysable, extensible and reusable.

3.2.1 Test architecture

The concept of the Test Architecture consists of defining what it is that is to be tested and what is to be used in performing the tests. One or more objects can be identified as the system under test. In addition test components can be identified; these are objects that are used within a test system to interact with the system under test or other components in order to perform the test.

3.2.2 Test behaviour

Certain behaviour is prescribed in a test which it is expected that the system fulfils. This behaviour needs to be expressed in a notation that allows it to be interpreted unambiguously in order to verify whether or not the test criteria have been fulfilled. Fortunately the existing notation and diagrams of the UML can be used to express the required test stimuli and observers, as well as to notate the co-ordination, interaction, sequence and stimuli needed to perform the tests and to express what is expected from them.

3.2.3 Test data

The data are the parameterised values that are actually used in the tests. In the context of the INESS project, the concept of test data is perhaps the least problematic test concept, as it has already been defined how this can be both created and used within the context of an executable UML interlocking specification.

Already in the Euro-Interlocking project, it was necessary to create test layouts on which the evolving functional requirements could be informally tried out and for this purpose a drag-and-drop GUI was created which allowed the user the possibility of creating a station layout of his or her choosing. In addition to this, it is also possible to create “control tables” in which the static data of each track element and route can be stored. Currently this data is entered and stored in MS Excel and is exported to the requirement simulator at run time. However, without extensive reworking, it should be possible for this data to be configured for use in testing too.

3.3 UML Testing Elements

Having defined the three high-level concepts of the testing profile we can now look at the elements used in each of them in detail.

3.3.1 Test Architecture Concepts

3.3.1.1 Test Context

Before any actual testing can be done, it is necessary to define a test context, which basically allows one to set the borders of what is to be tested. The test context covers both the test configuration and a set of test cases. Each test context (in the INESS project there is one) is related to a number of Systems Under Test (SUT) - again in the context of INESS there is only one: the interlocking. A test context also owns a potentially infinite number of test components and test cases. It can be said that these components are ‘owned’, as they have no valid meaning when divorced from their use in the test context. Again in the context of INESS, the number of test components will not be so large as is related to the number of interfaces of the SUT. The number of test cases however will be large and probably will be similar in number to the amount of use cases in the final requirements specification model.

This can all be shown graphically so:

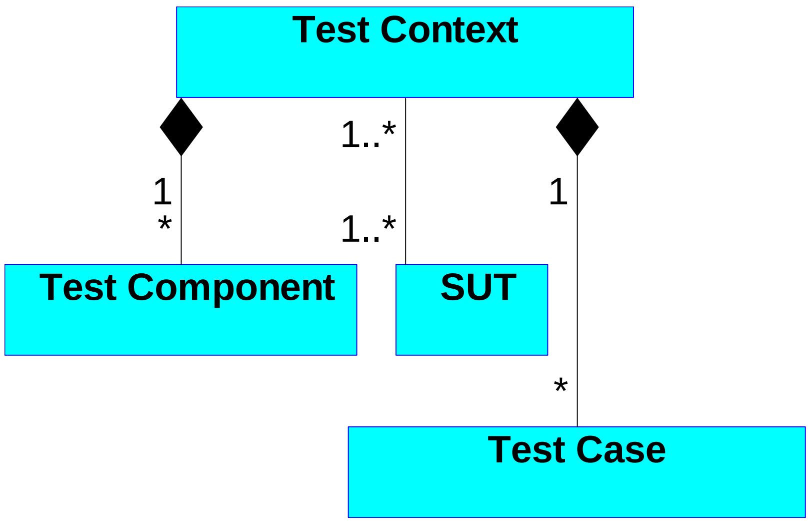

Figure 3

None of the elements shown above is entirely new to the modelling process, as each of them is related to a object in the UML meta model that is also used in the context of the requirements model. The Test Context is similar to the package concept in the UML meta model and the Test Case is a variation on a UML Use Case and both the SUT and the Test Component are forms of instance specifications. This underlines that using the UML Testing profile will not require the existing requirements modelling work to be lost, changed or stood on its head.

3.3.1.2. System Under Test (SUT)

Strictly speaking in the UML testing profile, the system under test is not part of the test model; it comprises a black box known only via the data it delivers at its external interfaces. In terms of a real interlocking undergoing factory acceptance testing, this may make some sense, but it needs to be kept in mind that at this stage of the INESS project, no boxes have been built and all that will be available for testing is the specification of a common core on which the interlocking system will be based. Thus although as much as possible of the system's functionality will be tested via the 'external object controllers', even these will only be *modelled* object controllers and it is unlikely that all the tests can be fully carried out without accepting a degree of glass-box testing.

In theory a SUT can be tested at different levels of abstraction – i.e. as a system *in toto* or as a collection of sub-systems – however in the context of INESS, unless the separate work stream on the functional apportionment of system architecture decides otherwise, the system specification will be viewed as one, indivisible system and it will be tested via its external interfaces via UML events and signals by the test components.

3.3.1.3 Test Component

A test component is any auxiliary element needed to carry out a test case, and is external to the SUT. It may have a set of interfaces through which it communicates with the SUT or with other test components. A test component may raise a stimuli or a sequence of stimuli against the SUT and/or it may passively receive stimuli from the SUT and process them. In the special case of a test arbiter, it may also interpret the stimuli from the SUT in order to appraise the success or not of the test being performed. In the context of INESS the test components will be defined as primarily the tester himself along with the various external object controllers which have been modelled outside the black box. In concrete terms this will require substitutes for the actual physical Object controllers to be created which will be designed for the purpose of receiving and reacting to the stimuli from the SUT (much as their ‘real’ physical counterpart) but which will also contain functionality to assist in the easy execution of the test – such as being configurable for degraded performance or even to be defective.

3.3.2 Test Behaviour Concepts

3.3.2.1 Test Objective

This concept is merely the high-level ‘business’ function which the system is being tested to check that it fulfils, this goal is reached through the use of a test case: the test objective describes what it is that the test case is intended to validate.

3.3.2.2 Test Case

The expected test behaviour is specified in a test case. This is the part of the test context which specifies how a set of co-operating components interact with the SUT to realise a test objective. A test case defines the behaviour which is realised by the set of test component objects. Like a test component, it too is ‘owned’ by a particular test context and is perhaps the most fundamental object in testing, as it acts as the container for specifying the desired behaviour of a test, including what to test, with which inputs and the expected result(s). A test case can be defined in terms of its pre- and post-conditions. In addition it is possible to specify sequences, alternatives, loops and stimuli to and observations from the SUT which are expected from the test. Furthermore, a test case may also invoke other test cases (a central part of the idea of reusability of the test specification) in verifying the adherence of the SUT to a test objective.

Each test case may have up to one pre-condition and one post-condition, both of which must be fulfilled at the appropriate point in time for the test to be considered successful. In addition the temporal aspects of a test case can be expressed via the test sequence diagram, in which any timing constraints or the above-mentioned iterations within the running of the test itself can be expressed. Thus, the graphic of the process can be extended as below:

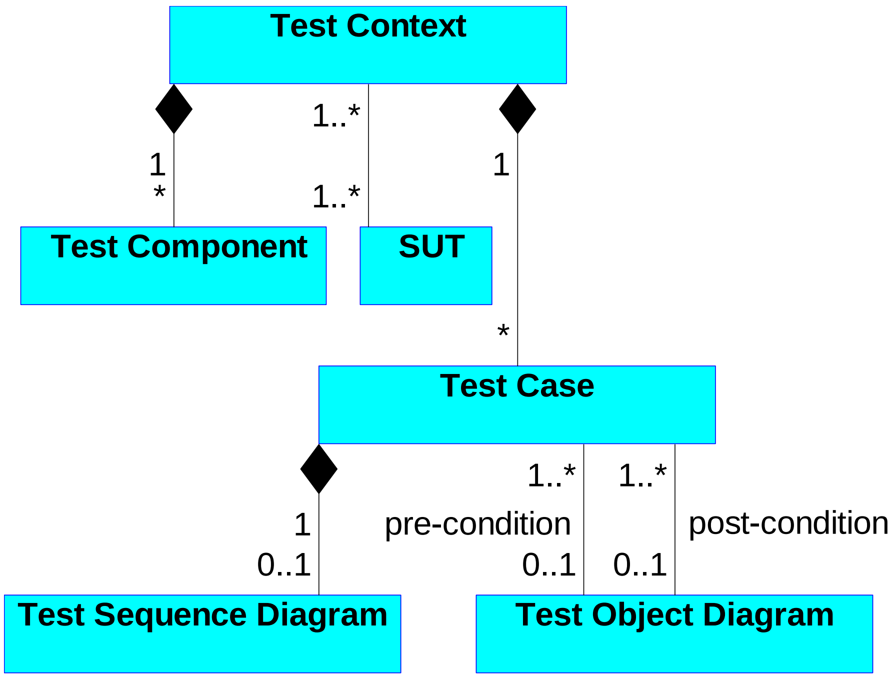

Figure 4

3.3.2.3 Arbiter

This is a role of a particular type of test component which is primarily responsible for assessing the outcome of the test and deciding whether or not the desired test criteria have been fulfilled. In itself it can be quite generically specified, as it is only at run time that it needs to be provided with the specific information about exactly what it is testing.

3.3.2.4 Test Data

The configuration is a pre-requisite for the creation of the simulation layout and both in order to avoid errors and to reduce the need for repeating work, it shall also be the basis for the data provided to the tester when populating the test cases with the data of which interlocking elements are to be tested. The mechanism by which this is to be done has not yet been defined, but is expected that it will largely be based on the previous work for configuring the layout. This part of the method will probably be defined last, as it will need to fit in with the nature and structure of the defined test method.

3.3.3 Method for Testing

Thus given both the outline of the UML testing profile, and the agreed inputs to this work stream, at a high-level, it is proposed to come up with a method along the following lines:

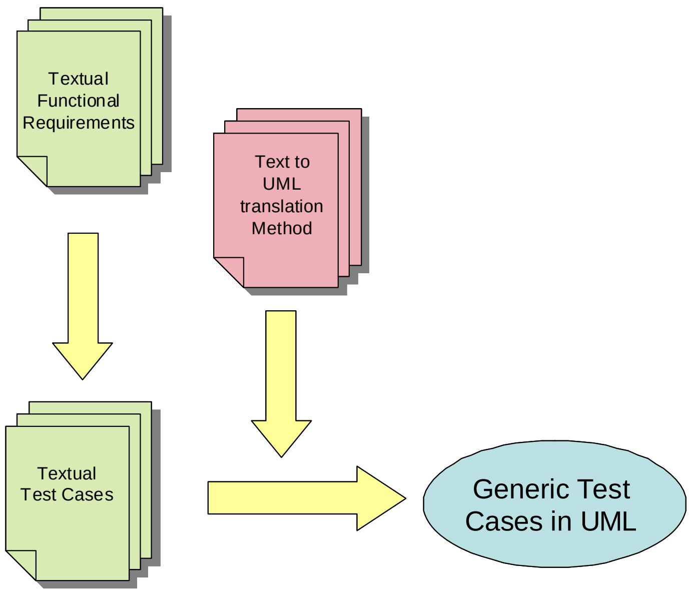
Textual Functional Requirements
Text to UML translation Method
Textual Test Cases
Generic Test Cases in UML

Figure 5

3.3.3.1 Modelling the tests

So far we have seen how the UML testing profile foresees that tests should be designed and conceived and now we shall move on to look at the details of expressing the actual tests in diagrammatic form. As has been mentioned above, as there are no entirely new diagram types used in the specifying tests in UML, it is hoped that this will simplify the acceptance of the method.

As we saw above in Figure 4, the test case is the receptacle for the details of the stimuli and behaviour for a given test and we shall now look in detail at each of its associated diagram types.

3.3.3.1.1 Pre/Post-condition diagrams

Both of these diagrams share the same basic form and are used to contain static configuration data which is required at either ‘end’ of the test. The diagrammatic form used for this is based on the UML Object Diagram, which we shall now look at via the example of how the static classes and the living object are displayed on their respective diagrams. Figure 6 below shows the abstract relationship between a “Lockable device” Class and the “Track” Class to which it is associated.

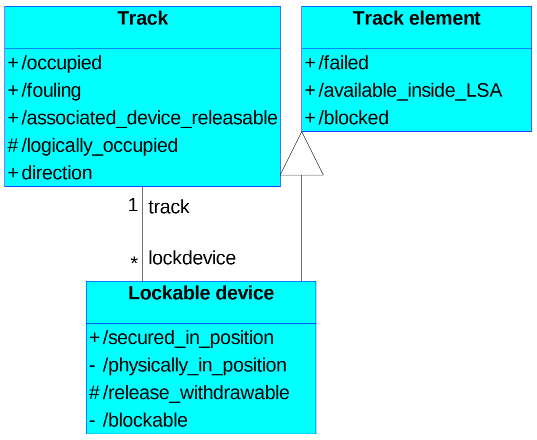

Figure 6

A possible Object Diagram of an instantiation of the above may look as follows:

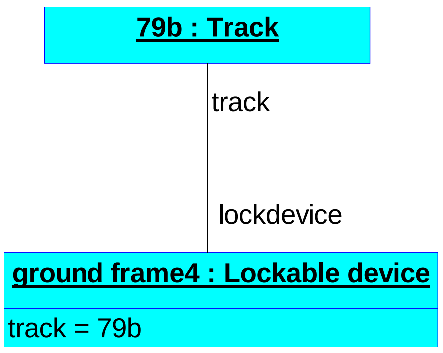

Figure 7

In this Object Diagram the Class names have been augmented with the names of the actual track and the lockable device instances in question. Furthermore, the multiplicity of the relationship between the groundframe and its track is now specified and the generalisation relationship from the Lockable device to the super class Track element has disappeared. This is because once an instance is created, there are no more super classes or sub classes, there are only instances and the distinction between the various classes from which an instance is created are no longer relevant. The relationship between the groundframe and 'its' track is also populated as a so-called 'slot value' on the object itself.

This is the diagram form which will be used for specifying the pre and post conditions of the objects being tested. However, if this were to be done uniformly for *all* instances of the simulation there would be several diagrams per object. This would quickly defeat the goal of creating a method that allows the test specification to be sufficiently generic as to allow it to be used for different layouts. Simply put, if testing every layout required a dedicated set of pre- and post-condition diagrams, the result could hardly be described as a method, but rather a specific catalogue of tests by hand, and thus we would not have achieved our goal.

Instead of looking at every pre- and post-condition for each instance, one could instead use the Object Diagrams to specify the *roles* played by the various objects and when one does this, it is expected that the total number of objects required to specify tests for a given layout falls dramatically and becomes much more manageable. The reason for this is twofold: firstly not all the statuses to be tested are available on the external object controller instances, which reduces the overall number of their unique combinations. In other words, if a physical point only knows whether it is ‘left’, or ‘right’ or ‘moving’, but not ‘trailed’, then any test requiring ‘trailed’ to be a pre- or a post-condition will not be able to be modelled in the physical point, but must be derived by some other method. The other reason is that by modelling generic roles which a given physical element can have instead of modelling of the actual states of all its instances, the number of diagrams needed is reduced enormously. It is only at run time that the instance roles and the actual instances would have to be brought together. To put it another way, it is only needed to show the roles a given object plays in diagrammatic form, *which* actual objects have the given role(s) can be defined later when the test is run. As these pre and the post-conditions are merely roles it may often be the case that what is a pre-condition of one test is also a post-condition in another. This too allows each instance specification to be used more than once, which also reduces the complexity of the whole – as the aim of this method is to be able to cover all the possible testable permutations of the instances in as few diagrams as possible. As noted, it is expected that this will reduce the total number of diagrams required, but this will not be confirmed until the actual test cases to be modelled are available from Work Package 4.2.

In the requirements model, the diagram form used for the representation of the entities of the specification is the Class Diagram, which displays the static qualities of each identified sub-component. The Object Diagram builds on this same basis, but with the difference, as the name suggests, that the entities displayed are no longer static classes, but ‘living’ instances; actual objects which have been created from the given class, rather than merely the abstract on which they are based. However, for the purposes of testing, merely creating and showing instances is not sufficient, as the aim is to specify how a given instance’s properties should differ (or not) at the end of a test from those at the start of the test. Thus it is necessary to be able to model instance *specifications*, which are analogous to a snapshot of the given instance at a given time. To give an example, the Object Diagram allows a particular point in a station layout to be identified and addressed; it allows the configurable properties of that point to be defined, but only once per instance, which excludes the possibility of showing how the object’s properties are to change over time (i.e. before and after the test). Thus, sensibly speaking, it must be possible to create multiple instance specifications per instance in order to fully document a test case.

Something else to keep in mind is that in a real factory acceptance test, the points in question (the ‘living’ instances mentioned above) would be actual physical objects, but as the scope of this study is not full end-to-end testing, but merely proving the specification, there are no physical points, merely simulations of the substitutes physical points. Conceptually this need not matter, as long as one is capable of grasping the various parallel levels of abstraction that this method necessitates.

Figure 8 below shows this part of the abstraction for pre- and post-condition diagrams. The point Class on the left is first specified in the persistent model. This class represents the interlocking's perception of the external point and contains its functionality. At run-time it is used as the basis for the creation of a number of given objects, all of which share the same common functionality as described in the point Class, but which differ according to the configuration values provided for each of them at instantiation. One physical point – or rather its simulated substitution is also created per abstract point instance as dictated by the given railway layout.

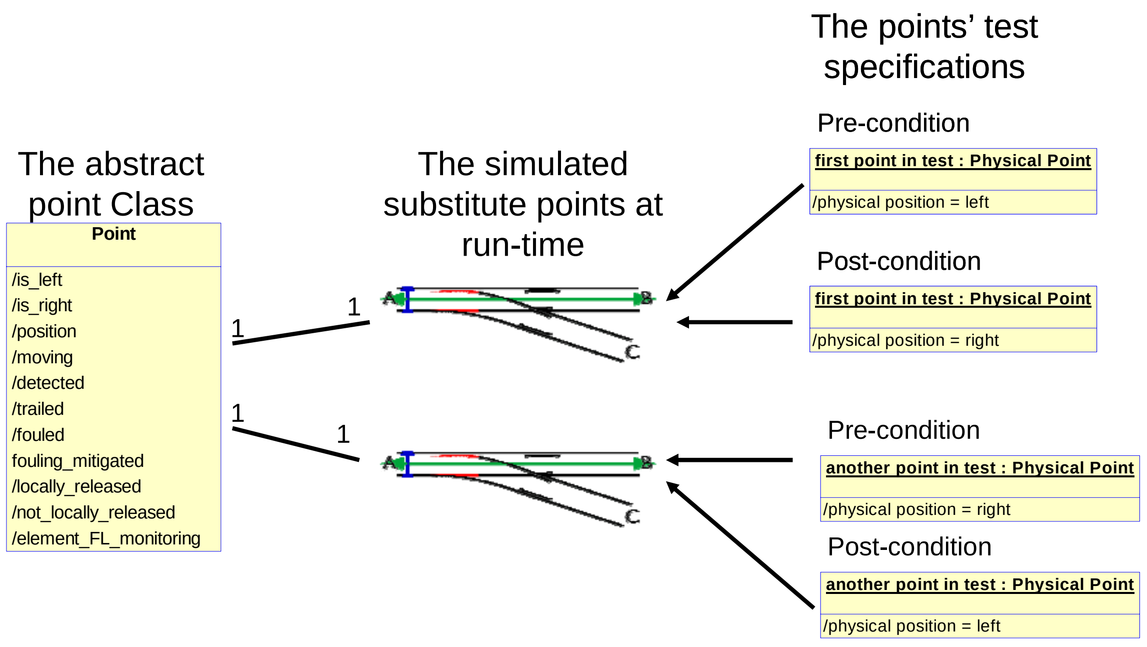
The points' test specifications
The abstract point Class
The simulated substitute points at run-time
Point
/is_left
/is_right
/position
/moving
/detected
/trailed
/fouled
fouling_mitigated
/locally_released
/not_locally_released
/element_FL_monitoring
1
1
1
1
A
B
C
Pre-condition
first point in test : Physical Point
/physical position = left
Post-condition
first point in test : Physical Point
/physical position = right
Pre-condition
another point in test : Physical Point
/physical position = right
Post-condition
another point in test : Physical Point
/physical position = left

Figure 8

It is important to note that although both the abstract and physical point instances are formal representations of the classes from which they have been instantiated, they are in themselves not specifications, but the temporary realisations of the specification, and as such within the context of INESS, they are not suitable as the basis for any persistent prescriptive behaviour. This is because we are striving to create generic test cases, rather than merely creating and observing the simulation's behaviour without checking it against any formal artefact. Thus, for this method to be useful, other persistent entities need to be created by which the behaviour of the instances can be checked at run time, and these are the instance specifications. These instance specifications are both persistent, in that they are modelled elements and descriptions of the required behaviour of particular roles which certain instances will need to fulfil in the given tests.

Thus we now have the persistent class and also the instance specifications (in this case being used as pre and post conditions) which can be used to observe the behaviour of the simulated substitute of the physical point at run-time. Because they are persistently modelled, they can also be re-used, by being linked to other instances of the physical point substitutes in other tests.

Having defined how to use the concept of the instance specification, the next step is to reduce the amount of redundancy necessary in each of them. For example, there will be many examples in which the point in question will be required to be not broken. To achieve a degree of efficiency in the modelling of this and to avoid unnecessary repetition, a mechanism has been added whereby a given instance specification can be refined by the values present in another.

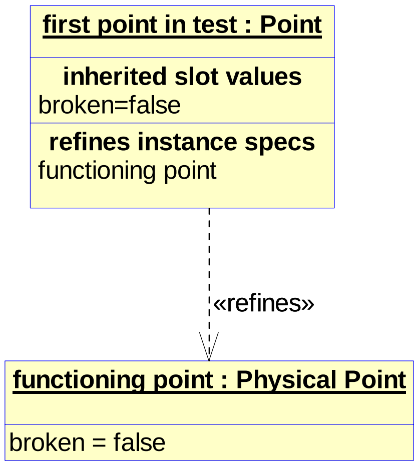

Figure 9

To do this, Artisan studio needs to be embellished with a profile that supports these features, and the objects will be shown in yellow in this and subsequent diagrams in order to distinguish them from the normal green object shown in the earlier diagrams. The real advantage of this method can be seen when multiple instance specifications are in use:

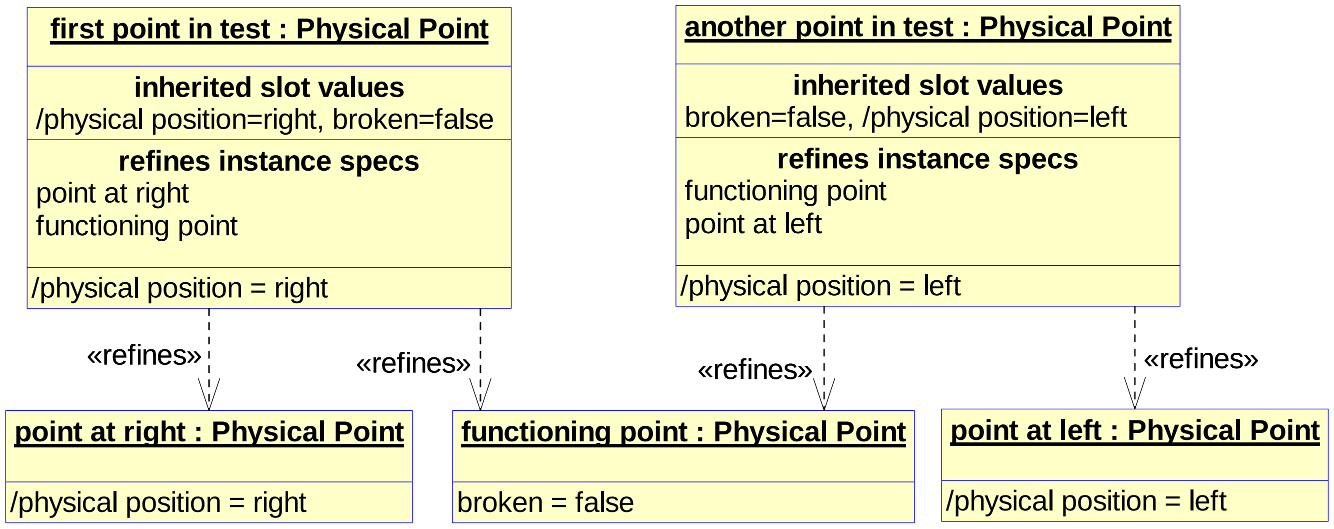
first point in test : Physical Point
inherited slot values
/physical position=right, broken=false
refines instance specs
point at right
functioning point
/physical position = right
another point in test : Physical Point
inherited slot values
broken=false, /physical position=left
refines instance specs
functioning point
point at left
/physical position = left
«refines»
«refines»
«refines»
«refines»
point at right : Physical Point
/physical position = right
functioning point : Physical Point
broken = false
point at left : Physical Point
/physical position = left

Figure 10

Each subsequent instance specification requiring that the point be in the left position or not broken can merely be refined by associating it to the relevant instance specification. In the example below numerous slot values are shown as being inherited from the 'slow functioning point' instance

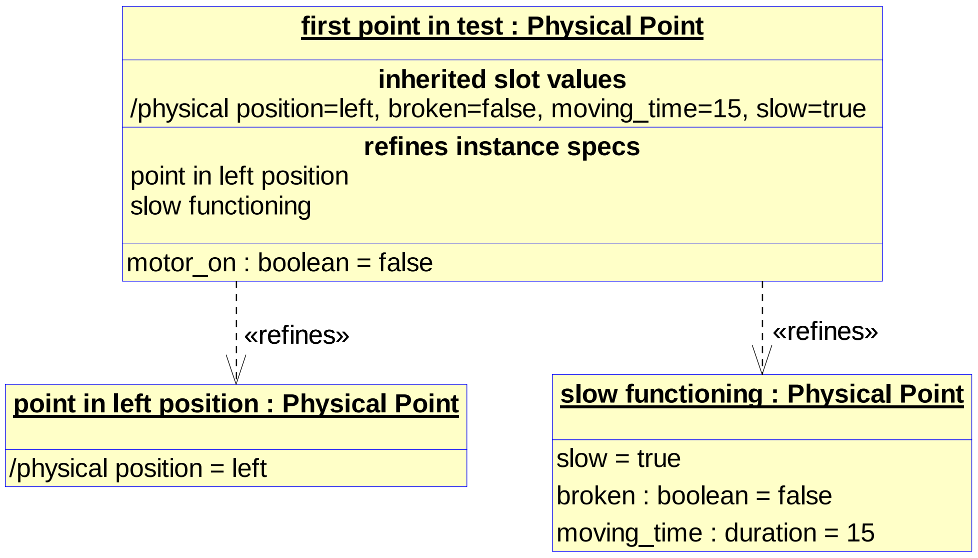
first point in test : Physical Point
inherited slot values
/physical position=left, broken=false, moving_time=15, slow=true
refines instance specs
point in left position
slow functioning
motor_on : boolean = false
«refines»
«refines»
point in left position : Physical Point
/physical position = left
slow functioning : Physical Point
slow = true
broken : boolean = false
moving_time : duration = 15

Figure 11

The names of the instance specifications which are used to refine the points in the test show that they generally fulfil a physical role, such as 'functioning' or 'at left'. However the 'refined' instances are named differently. In the example above the name given is 'first point in test', which may sound somewhat vague. The reason for this, as mentioned above, is that these diagrams are only used to specify the conditions expected of the actual instances in the test and they are not the instances themselves. Thus a mechanism is still needed to allow the instance specification to be connected to the actual physical point that they have been chosen to test and this is done via parameterisation.

3.3.3.1.2 Test Maps and Test Cases

An integral part of defining generic tests is to create a hierarchy by which the actual tests to be run can be related to one another in such a manner as to promote efficiency and the general avoidance of redundancy. This can be achieved by defining how one given test case can invoke another. This enables the defined test cases to be fairly small and simple, so that they can be more easily related via the 'include' relationship to other test cases.

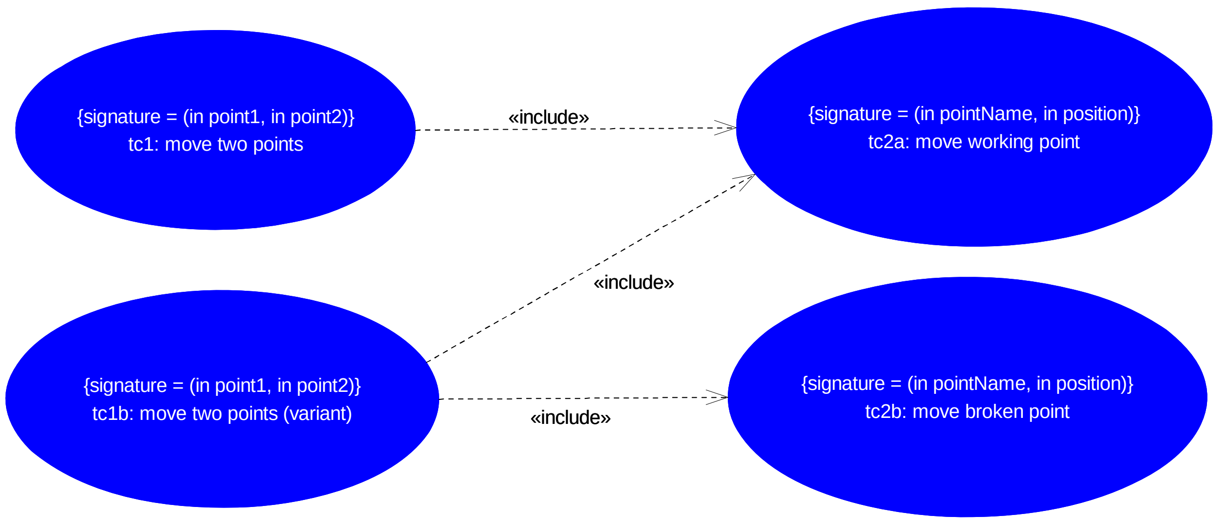
{signature = (in point1, in point2)}
tc1: move two points
«include»
{signature = (in pointName, in position)}
tc2a: move working point
«include»
{signature = (in point1, in point2)}
tc1b: move two points (variant)
«include»
{signature = (in pointName, in position)}
tc2b: move broken point

Figure 12

Figure 12 provides an overview of two simple use cases (on the right) and how they can be invoked by other, more complex test cases (on the left). In the concrete case given above, there is one test case each for moving a correctly-functioning point and a broken point and these tests can be invoked by other tests which either test two working points, two broken points or one of each.

The key to the invocation of one test case by another is the "includes" relationship and the parameters which it provides. The 'formal parameters' for the generic test case are declared at the level of the persistent, specified test case, and in the case of tc1, they are 'point1' and 'point2'. At run-time these will be populated with the 'actual parameters', which in this case will be the instances of the points being tested.

3.3.3.1.3 Test Sequence diagrams

As we saw in Figure 4, a test case may contain a sequence diagram which is used to express the dynamic behaviour of a test case and which can also be used to specify how the parameters will be determined and conveyed to any 'included' test case. At run-time the two temporary variables (X and Y) will be populated with the names of the points under test and these will be forwarded via the include probe to the test case tc2a, which will try to move two functioning points, one to the left and the other to the right.

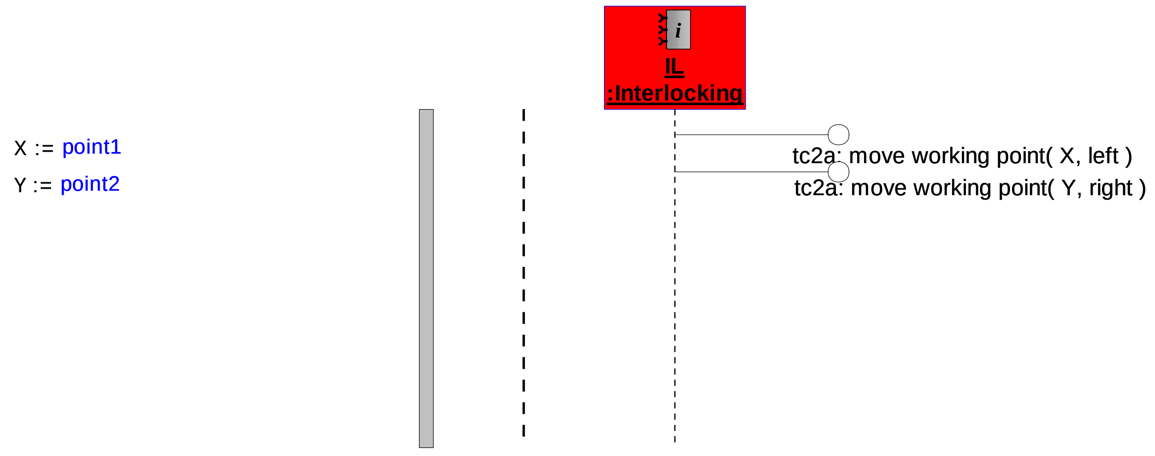
IL
:Interlocking
X := point1
Y := point2
tc2a: move working point(X, left)
tc2a: move working point(Y, right)

Figure 13

The test sequence diagram above functions more or less only as a vessel to collect and forward the requests to the points in questions, but the test sequence diagrams which it invokes are somewhat more complicated.

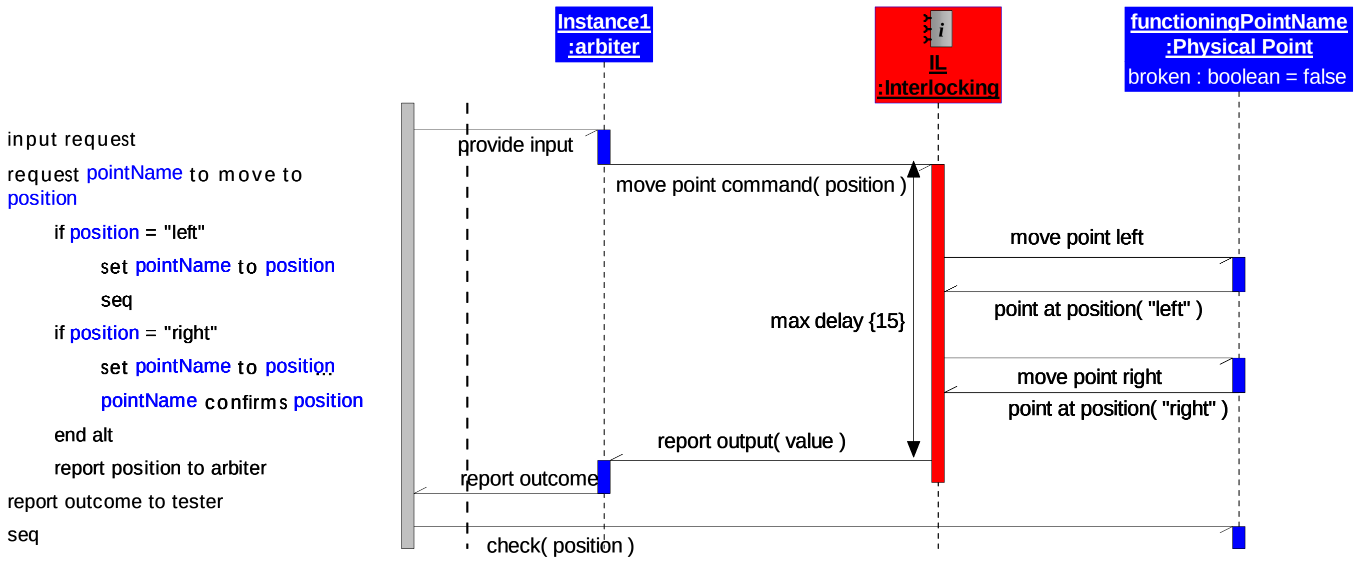

Figure 14

In Figure 14, regardless of whether the test to move a single functioning point is addressed directly by a tester or whether it is invoked by another test case, it will require parameterised input as to which point is being moved and to which position.

For this to happen a generic event provide input event is sent from the system boundary to an instance of a test arbiter. This is a specific instance of a generic test component that knows the interface specification of the interlocking and can create and forward it the required event along with the appropriate parameter value(s) to initiate the SUT for the purposes of running the test. In this particular case it does so by sending a particular instance of point within in the interlocking a move point command which is qualified with the actual position requested. In addition to this it stores the information it has sent in order to evaluate the success of the test once the interlocking reacts.

What follows is the behaviour as specified in the requirements model according to the parameters invoked in the test case. The events used are those of the given interface specifications. The result should be that the test component which is emulating the functioning physical point should report back to the interlocking that the point has moved to the requested position.

Once this has happened the interlocking reports this back to the same instance of the arbiter as forwarded it the move request in the first place and this arbiter then decides whether the test has succeeded or not. Finally the tester can also request a status report directly from the physical point emulator to see if its report position complies with that which the interlocking is reporting for it.

In addition to all this, there is also a timing restraint, by which it is specified that the interlocking must report back the detection of the point in the correct position within 15 seconds. Failure to do so would also result in the test failing.

3.3.3.1.4 “Negative” testing

Lastly, although most requirements prescribe that something should happen (the point shall move left); equally a large number of requirements specify that at no stage during a given process should something else happen. This behaviour can also be specified in test cases, but slightly differently than for the ‘positive’ tests. The reason for this is that whereas the ‘positive’ test cases care that something particular happens at a given time, these ‘negative’ test cases specify that at no time during the running of test should such and such occur. These ‘negative test case observers’ can have state diagrams which look like this:

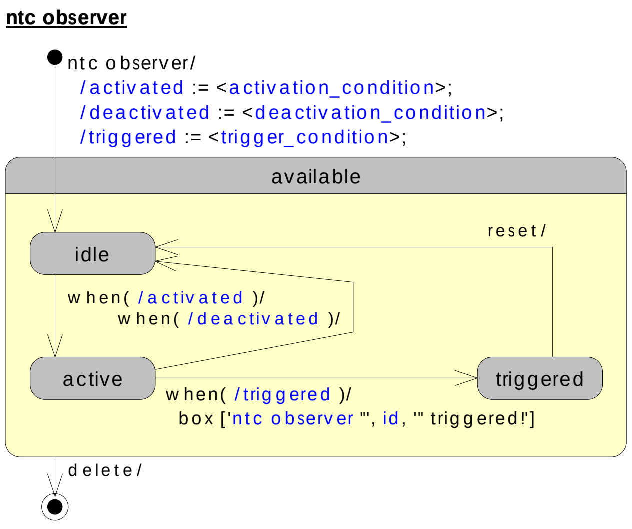
ntc observer
ntc observer/
/activated := <activation_condition>;
/deactivated := <deactivation_condition>;
/triggered := <trigger_condition>;
available
idle
reset/
when(/activated)/
when(/deactivated)/
active
triggered
when(/triggered)/
box ['ntc observer '", id, '" triggered!']
delete/

Figure 15

Such an entity would be highly generic and fully configurable in terms of what it looks for, where it looks and in what circumstances. Additionally it can also be specified how such a test component should react if it observes behaviour which has been defined as its trigger. Such test components could be invoked by a number of test cases and their behaviour specified as required:

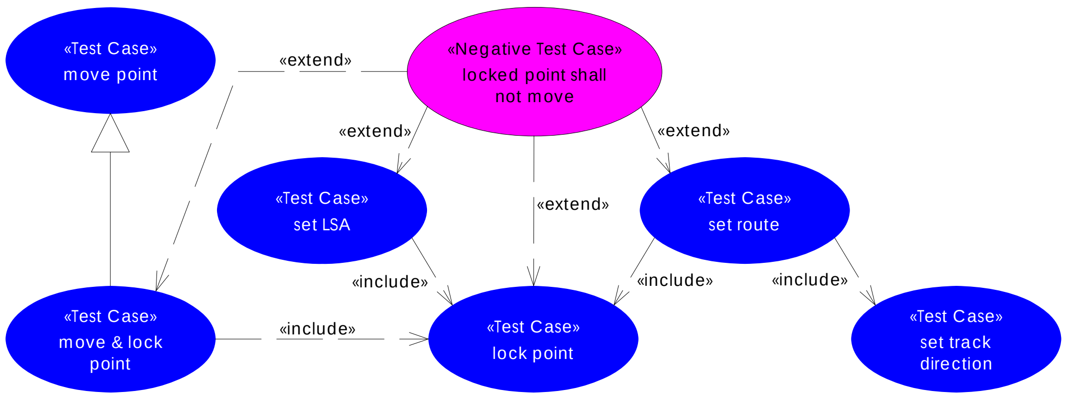

Figure 16

Section 4 – CONCLUSIONS

Exactly how much or how little of the above-defined method will be required will only be known once the actual test cases are available. It is unlikely that nothing changes as fundamentally it will be the actual test cases which will dictate the direction of the test method. But in general it is hoped that this preliminary paper will serve to stimulate discussion about what is expected from specification testing as a concept.

One clear advantage with this approach is that although some work remains to be done to actually populate the links between an instance specification roles and the actual elements to be used to support the test, the information itself is already available to the simulator – as the configuration data is used in setting up the simulation in the first place.

The next steps from here will be to add actual data to the test cases as shown here and to define a process for the running of the tests, the evaluation of the results, the issue of mandatory and optional tests as well as the production of the test case report documentation.

Section 5 – BIBLIOGRAPHY

Object Management Group, *UML 2.0 Testing Profile*, July 2005.

S. Blom, N. Ioustinova, J. v. d. Pol, A. Rennoch, N. Sidorova, *Simulated Time for Testing Railway Interlockings with TTCN-3*.

A. Knapp, S. Merz, and C. Rauh, *Model Checking Timed UML State Machines and Collaborations*, FTRTFT 2002 : formal techniques in real-time and fault-tolerant systems, Oldenburg, 9-12 September 2002.

S. Bardin, *Vers un Model Checking avec Accélération Plate des Systèmes Hétérogènes*, PhD thesis, École Normale Supérieure de Cachan, October 2005.

Zhen Ru Dai, *Model-Driven Testing with UML 2.0*.

P. Baker, Zhen Ru Dai, J. Grabowski, Ø. Haugen, S. Lucio, E. Samuelsson, I. Schieferdecker, and C.-E. Williams, *The UML 2.0 Testing Profile*, 2004.

S. Gruner, *From Use-Cases to Test-Cases via Meta-Model-based reasoning*, ICFEM-affiliated Workshop UML+FM'08 on the Unified Modeling Language and Formal Methods, Kitakyushu (JP). Published in Innovations in Systems and Software Engineering: A NASA Journal, Vol.4 No.3, pp.223-231, Springer-Verlag, October 2008.

W.T Tsai, R. Paul, Zhibin Cao, Bingnan Xiao, Lian Yu, *Adaptive Scenario-Based Testing Using UML*, October 2002.

K. Bisgaard Lassen, S. Tjell, *Model-based requirements analysis for reactive systems with UML sequence diagrams and coloured Petri nets*, Innovations in Systems and Software Engineering, volume 4, number 3, October 2008.

Xiaoshan Li, Dan Li, Jicong Liu and Zhiming Liu, *Validation of Requirement Models by Automatic Prototyping*, UML&FM08, October 2008, Japan.

C. Sibertin-blanc, N. Hameurlain, O. Tahir, *Ambiguity and Structural Properties of Basic Sequence Diagrams*, UML & FM Workshop, Kitakyushu, Japan, 27/10/2008.

J. Bézivin, *Do model transformations solve all the problems?*, International Conference on Formal Engineering Methods, Kitakyuschu City, October 2008.

J.-M. Bruel, *Why did I give up doing UML+Z (lessons learned)?*, UML & FM Workshop, October 2008.

I. Schieferdecker, J. Grabowski, *The UML 2.0 Testing Profile*, TTCN-3 User Conference, 3-5 May 2004, Sophia-Antipolis, France.

B. Powel Douglass, *Model-Based Testing Using UML and Test-Driven Development*, IBM Rational Software Conference 2009, 4th November 2009.

E. Samuelsson, *Graphical Testing With UML The UML Testing Profile*, Telelogic presentation.

Section 6 – ANNEXES

The UML Testing Profile is a modelling language allowing the specification of tests. It is based on the UML diagrams and concepts. It can be used to describe the test for technical systems, and can be applied to various domains. It can of course be combined with UML.

The formally released version of UML TP version 1.0 can be downloaded at the following address (.pdf file): http://www.omg.org/spec/UTP/1.0/PDF/