

GENERIC ARBITRATIONS FOR TEST REPORTING

Joanna Chimiak–Opoka and Sarah Löw
and Michael Felderer and Ruth Breu
Institute of Computer Science
University of Innsbruck, Austria
joanna.opoka@uibk.ac.at

Frank Fiedler and Felix Schupp
softmethod GmbH
Munich, Germany
felix.schupp@softmethod.de

Michael Breu
arctis software technology GmbH
Inzing, Austria
michael.breu@arctis.at

ABSTRACT

Generic arbitrations over multiple test runs are important from an industrial point of view as a mean to provide a better understanding of the testing process, to enable monitoring of implementation progress and to estimate an overall software system reliability and to reduce the development cost. We propose to combine behavioural and performance aspects into uniform validation of a system under test which is especially important in the domain of telecommunication, real–time databases and concurrent systems. We introduce expressive and extensible definitions of verdict functions and show how they should be integrated into arbitrations and test reports. The work presented in this paper is a part of the Telling TestStories framework dedicated to model–driven testing in early development stages.

KEY WORDS

Quality Assurance, Software Testing, Verdict Functions, Arbitrations, Test Reporting.

1 Introduction

The increasing size of software systems triggers off new challenges. Because of the large size of models it becomes hardly feasible to get a in–depth understanding of a whole system at a low abstraction level. Methods for simplification become necessary to retrieve information from a system. Models and metrics have been proposed to alleviate the complexity problem. In this context arbitrations, consolidating test evaluation results, and reporting can help to gain a *better understanding of the testing process* shifting it to a more general level.

Another important issue for arbitrations and test reporting is the *monitoring of implementation progress*. Defining well designed system tests, may result in a statement of the overall work progress and can early help pointing out the system being behind a schedule.

Moreover, in some areas arbitrations are the only mean for test evaluation. Many software systems under development have to fulfil service level agreements. These can be very strict or fuzzy agreements. In order to prove the system fulfilling the requirements several tests must be run to state an *overall system reliability* (examples in Section 1.1).

Contractual specifications of large IT systems usually contain (informal) functional tests, reliability and performance requirements for these systems. State of practice is that these requirements are typically managed manually, i.e., functional tests are executed manually according to a commonly agreed test script. Reliability and performance tests are mainly part of the final acceptance procedures. The effort for manual testing is extremely high and may cover up to 50% of the development costs. Even partial automation of test execution and arbitration can cause significant *reduction of the development costs*.

In our cooperation project we develop the framework Telling TestStories [1] supporting tests at early stages of system development. Telling TestStories has been designed based on industrial requirements in the areas of telecommunication and transportation systems. Our framework focuses on elicitation and validation of requirements for service oriented systems. The validation is done at the system and acceptance testing level. Arbitrations and reports constitute an important part of the framework and they are in the focus of this paper.

1.1 Industrial needs

In our cooperation projects, arbitrations and reports are requested in the area of telecommunication, real–time databases and concurrent systems. In this section we present business cases pointing out industrial needs.

In the area of **telecommunication** the software system has to fulfil strict requirements. Imagine a software system reacting to incoming calls. Dependent on a caller's number, the desired service has to calculate some parameters and decide to which destination number the incoming call has to be routed. There are two kinds of requirements to the services related to behaviour and performance. Firstly, a correct implementation must be assured, i.e., the correct number must be selected. Secondly, the number must be selected as fast as possible, since otherwise the person calling might hang–up the call even before having been routed. In this example the behavioural requirements are essential but without assuring high performance they become useless. In a case when the correct number is provided too late, the service has to exit and route the call to a predefined default number, in order not to loose the calling customer.

In the domain of **real-time database systems** a lot of information is stored. As an example we will consider a database with many personal data records. At the service agreement level we can have, apart from behavioural requirements, the following performance requirement: queries to the database have to provide the result within one second in 99% of all cases, and within less than three seconds in the remaining cases. It is not possible to decide if the system really fulfils the service agreement based on results of single test runs. In such situation we need arbitrations over sequences of test runs to obtain a reliable decision on the system performance.

In case of capacity of **concurrent systems** less restrictive arbitrations are required. If a maximum of concurrent users a system can handle is equal to 80, we would need a verdict of a test scenario, that out of 100 parallel login requests, at least 80 requests must succeed, whereas the others may fail explicitly or stay inconclusive.

The aforementioned issues show the need for verdicts combining behavioural and performance aspects and evaluated over sequences of test runs. Additionally reporting is requested to summarise the overall performance and correctness of the system. Reports give an overview for a development team, as they show progress of the development process and enable effort prediction. Reports are also an important source of information for the ordering customer who wants to be kept informed about the project's progress. In the next section we show work related to arbitrations and reporting.

1.2 State of the art

Regarding arbitrations some standardisation and formalisation proposals appeared in recent years. There are two important **standards for model driven testing** which consider basic arbitration strategies, namely the Testing and Test Control Notation Version 3 (TTCN-3) [2] and the UML 2.0 Testing Profile (U2TP) [3]. Both standards provide domain specific modelling languages for test specification and evaluation with a set of possible **verdict types and an order** over them. They define some basic arbitration strategies and offer user defined strategies, but give no further details on this topic. In our approach we rely on a subset of the verdict types defined in the standards, which can be easily extended by other types (see Section 3.4). We go further than the standards as we give precise definitions of the verdict functions of different types.

An important work in the field of **arbitration formalisation** is [4], where a function returning a verdict over a set of observations is defined. This work gives formal background for behaviour based verdicts, introduces formal order and properties of different verdict types. The functions defined in [4] do not cover timing aspects, whereas in our approach we define verdict functions for behaviour and performance in a pragmatic and uniform manner.

Complex arbitrations calculated over verdict functions can be seen as metrics over test runs. They are im-

portant from the practical point of view as they enable estimation of a current progress and predict future test efforts [5]. The test progress monitoring can be achieved by **test reporting**. Among tools supporting creation of reports we can distinguish between general purpose tools, such as BIRT¹, or tools dedicated to test reports, such as open source Testopia² or the commercial spiraTest³. General purpose tools offer flexibility, but need a lot of tailoring to be used in test reporting. Test report tools are often integrated into complex test environments and support customisation of reports at the level of single test runs.

1.3 Our contribution

The proposed framework for generic arbitrations and reporting has the following properties:

- capability of *tests results aggregation*, which is useful for the current state of a project and a progress overview at the project management level that can be integrated with project *reporting*,
- integration of *behavioural and performance testing* into one arbitration system which is important for real time system testing,
- a *flexible and expressive arbitration system* based on verdict functions and logical expressions which can be easily tailored to particular needs and extended with additional verdict types or verdict functions.

1.4 Structure

In the next section we introduce the basic terminology. In Sections 3 and 4 we define behavioural and performance verdict functions, respectively. Section 5 presents the way generic arbitrations can be integrated into test reports. The last section provides the conclusion and points out future work.

2 Terminology

In this section the basic terminology used in our proposal is introduced. The relations between concepts are depicted in Fig. 1. The bottom level corresponds to test specification, the next upper level to test execution, the further one to test evaluation and the top levels to test arbitration and reporting.

At the *test specification level* we have the concept of a test case which is defined as follows:

Test Case is a set of conditions or variables under which a tester will determine if a requirement or use case upon an application is partially or fully satisfied. A *Test Case* contains enough information to be executed as a \rightarrow *Test Run*.

¹BIRT, <http://www.eclipse.org/birt/>

²Testopia is a test case management extension for Bugzilla, <http://www.mozilla.org/projects/testopia/>

³spiraTest, <http://www.inflectra.com/SpiraTest/>

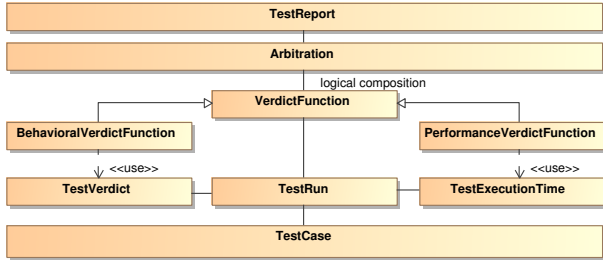


Figure 1. Basic concepts and relations between them

At the *test execution level* we have three concepts which are related to test execution and its monitoring:

Test Run is the execution of one \rightarrow Test Case.

Test Verdict is a judgement upon the result of a single \rightarrow Test Run. We consider the following verdict types (defined in the same way as in [3], they can be easily extended by remaining types, such as none or error introduced in [2, 3]):

- pass indicates that the test behaviour gives evidence for correctness of the system under test (SUT) for the specific \rightarrow Test Case,
- fail describes that the purpose of the \rightarrow Test Case has been violated,
- inconc (inconclusive) is used for cases where neither a pass nor a fail can be given.

Test Execution Time is the time needed to execute a single \rightarrow Test Run.

The next three concepts are related to the *test evaluation level* and are defined as follows:

Verdict Function returns an encoded value which is based on a single \rightarrow Test Run or on a sequence of \rightarrow Test Runs. We consider two types of Verdict Functions: \rightarrow Behavioural Verdict Function and \rightarrow Performance Verdict Function.

Behavioural Verdict Function returns a result based on the \rightarrow Test Verdicts of \rightarrow Test Runs. Section 3 is dedicated to this concept.

Performance Verdict Function returns a result based on the \rightarrow Test Execution Time of \rightarrow Test Runs. Section 4 is dedicated to this concept.

The last concepts belong to the *arbitration and reporting levels* and are defined as follows:

Arbitration is a decision on the highest abstraction level whether a sequence of \rightarrow Test Runs satisfies a given arbitration criteria. It uses logical and arithmetical operators to combine results of \rightarrow verdict functions. See Sections 3.3, 3.4, 4.3, 5.1, and 5.2 for more details.

Test Report summarises \rightarrow arbitrations of selected \rightarrow test runs. Section 5.3 is dedicated to this concept.

3 Behavioural verdict functions

As mentioned in the previous section, behavioural verdict functions are related to the expected behaviour of a SUT. In this section we describe how behavioural functions are calculated for a single test run (Section 3.1) and for a sequence of test runs (Section 3.2). We give some examples and demonstrate how an arbitration can be built upon behavioural verdict functions (Section 3.3). Further we demonstrate in which way arbitrations defined in the TTCN-3 standard can be expressed using our definitions (Section 3.4).

3.1 Behavioural verdict functions for a single test run

At first let us consider a **single test run** tr , which represents an executed test case tc (as given in Fig. 1) and the corresponding test verdict v obtained for the test case tc . The verdict equals to one of the values mentioned in Section 2, $v \in \{\text{pass}, \text{inconc}, \text{fail}\}$. A test run is a tuple defined as:

$$tr = \langle tc, v \rangle. \quad (1)$$

Example 1 Consider a test that queries the SUT to find a user with a particular family name. If we take a concrete family name and define an expected result, e.g. a user id, we get a test case (tc_0). If we run tc_0 we get a test run tr_0 resulting in a verdict (e.g., $v_0 = \text{pass}$, if the expected result and the return value are equal): $tr_0 = \langle tc_0, v_0 \rangle = \langle tc_0, \text{pass} \rangle$.

Now we define a **behavioural verdict function for a single test run** $\mathcal{B}(tr)$, which returns an encoding of the obtained verdict:

$$\mathcal{B}(tr) = \begin{cases} \langle 1, 0, 0 \rangle & \text{if } v = \text{pass} \\ \langle 0, 1, 0 \rangle & \text{if } v = \text{inconc} \\ \langle 0, 0, 1 \rangle & \text{if } v = \text{fail} \end{cases} \quad (2)$$

Verdicts are encoded as unit vectors in a vector space whose length corresponds to the number of different verdicts. We introduce this encoding to enable an easier manipulation (e.g. projection) and implementation.

Example 2 For the test run tr_0 from the previous example, we get $\mathcal{B}(tr_0) = \langle 1, 0, 0 \rangle$.

The positions of the verdict types in the unit base correspond to the definition of a total order on the set of all verdict types, $\text{pass} > \text{inconc} > \text{fail}$. This is a shorthand notation for the following structure: whenever we obtain a verdict v , we ignore all verdicts v' for which we have $v' > v$. In other words if we have inconc , we ignore pass , if we have fail , we ignore pass and inconc . The order was introduced to enable an easier implementation and it is compatible with orders introduced in [2, 3]. In TTCN-3 [2] the following verdict types are defined: none>pass>inconclusive>fail>error.

In U2TP [3] the verdict types and the order are the same only none is not considered.

To access a **value in the unit base** we introduce the projection operator π , e.g. to access the value for pass verdict we use $\pi_{\text{pass}}(\mathcal{B}(tr))$. Additionally we define $\pi_{\text{all}}(\mathcal{B}(tr))$ to obtain the sum for all types:

$$\pi_{\text{all}}(\mathcal{B}(tr)) = \sum_{type \in \{\text{pass}, \text{inconc}, \text{fail}\}} \pi_{type}(\mathcal{B}(tr)). \quad (3)$$

Example 3 For tr_0 defined in Example 1 we obtain:

$$\begin{aligned} \pi_{\text{pass}}(\mathcal{B}(tr_0)) &= 1, & \pi_{\text{inconc}}(\mathcal{B}(tr_0)) &= 0, \\ \pi_{\text{fail}}(\mathcal{B}(tr_0)) &= 0, & \pi_{\text{all}}(\mathcal{B}(tr_0)) &= 1. \end{aligned}$$

3.2 Behavioural verdict functions for a sequence of test runs

Now we consider a **sequence of test runs** TR as a collection of single test runs. Based on the definition of a test run given in (1) we define TR as follows:

$$TR = \langle tr \mid P(tr) \rangle, \quad (4)$$

where P is a predicate determining a selection criterion. Depending on the selection criterion the collection can represent either test cases related to a single family of test cases built over a parametrised test case, or any desired collection of test cases which represents a part of system functionality. As the collection represents aggregated test runs it enables interpretation of their execution results at a more abstract level.

Example 4 We define TR_0 as a sequence of test runs that search different users. Assuming we executed three test runs, we can obtain $TR_0 = \langle \langle tc_0, \text{pass} \rangle, \langle tc_1, \text{pass} \rangle, \langle tc_2, \text{inconc} \rangle \rangle$ where tc_0 is defined as in Example 1 and tc_1, tc_2 are defined as tc_0 , but with different test data.

Now we will define **behavioural verdict functions for a sequence of test runs** $\mathcal{B}(TR)$. We have to extend the definition given in (2) as the function representing a sequence of number of occurrences of particular verdict types obtained for all test runs in the sequence:

$$\mathcal{B}(TR) = \langle \pi_{\text{pass}}(\mathcal{B}(TR)), \pi_{\text{inconc}}(\mathcal{B}(TR)), \pi_{\text{fail}}(\mathcal{B}(TR)) \rangle, \quad (5)$$

where

$$\pi_{\text{pass}}(\mathcal{B}(TR)) = \sum_{tr \in TR} \pi_{\text{pass}}(\mathcal{B}(tr))$$

and by analogy we define projections for `inconc` and `fail`.

Example 5 For TR_0 defined in the previous example, we may obtain $\mathcal{B}(TR_0) = \langle 2, 1, 0 \rangle$.

3.3 Arbitrations over behavioural verdict functions

To define the ratio of test runs with particular verdict types we use projections:

$$\pi_{\text{pass}}^{\%}(\mathcal{B}(TR)) = \frac{\pi_{\text{pass}}(\mathcal{B}(TR))}{\pi_{\text{all}}(\mathcal{B}(TR))} \quad (6)$$

and by analogy we define projections for `inconc` and `fail`. Using percentages we can define **arbitrations** over verdicts. For example, we can express upper or lower bounds for test verdicts.

Example 6 We can define that at least three quarters of all test runs must obtain a pass verdict, which can be expressed as $\pi_{\text{pass}}^{\%}(\mathcal{B}(TR)) \geq 75\%$. Considering this criteria TR_0 from Example 5 is not satisfying the arbitration, as $\pi_{\text{pass}}^{\%}(\mathcal{B}(TR_0)) = 66\%$.

Moreover we can obtain more complex arbitrations by combining constraints with the **logical operators** *and* (\wedge), *or* (\vee) and *not* (\neg).

Example 7 With logical operators we can define that at least three quarters of all test runs must obtain a pass verdict and no more than 5% may obtain an inconclusive verdict, which can be expressed as $\pi_{\text{pass}}^{\%}(\mathcal{B}(TR)) \geq 75\% \wedge \pi_{\text{inconc}}^{\%}(\mathcal{B}(TR)) \leq 5\%$. Considering this criteria TR_0 from Example 5 is not satisfying the arbitration, as $\pi_{\text{pass}}^{\%}(\mathcal{B}(TR_0)) = 66\%$ and $\pi_{\text{inconc}}^{\%}(\mathcal{B}(TR_0)) = 33\%$.

3.4 Expressing TTCN-3 arbitrations

The definitions we proposed can be used to simulate concepts provided by TTCN-3, where every component computes a local verdict for a test case which is then combined by the arbiter to a global verdict. The default arbitration from TTCN-3 can be expressed as follows (for completeness `none` and `error` can easily be added):

$$TTCN(TR) = \begin{cases} \text{fail} & \text{if } \pi_{\text{fail}}^{\%}(\mathcal{B}(TR)) > 0 \\ \text{inconc} & \text{if } \pi_{\text{fail}}^{\%}(\mathcal{B}(TR)) = 0 \\ & \wedge \pi_{\text{inconc}}^{\%}(\mathcal{B}(TR)) > 0 \\ \text{pass} & \text{otherwise} \end{cases}$$

Example 8 Considering TR_0 from Example 4 we obtain $TTCN(TR_0) = \text{inconc}$.

The TTCN-3 standard mentions that users can define their own arbitration scheme, e.g. for performance tests, but no further details are given. In the next section we show how the performance aspect can be added to our formalism.

4 Performance verdict functions

As mentioned in Section 2, performance verdict functions are related to the test execution time, which is an important aspect for industrial applications. In this section we

describe how performance verdict functions are calculated for a single test run (Section 4.1) and for a sequence of test runs (Section 4.2). We give some examples and show how arbitrations can be defined based on both types of verdict functions (Section 4.3).

4.1 Performance verdict functions for a single test run

To handle performance constraints we extend the definition of a **test run**, given in (1), to the following triple:

$$tr = \langle tc, v, t \rangle, \quad (7)$$

where t is time needed to execute a test case tc obtaining the verdict v . The time is defined over natural numbers, representing time in a given unit (e.g. milliseconds), extended with a symbol for undefined time (\perp), in case of a time-out occurrence and the $\#$ symbol for not relevant results, in case when a projection returns no value. Hence we get $t \in \mathbb{N} \cup \{\perp, \#\}$. We introduce the following semantics of additional symbols: \perp is treated as positive infinite in comparisons, and $\#$ is omitted in arithmetical operations.

Example 9 Example 1 including test execution time can be represented as $tr_0 = \langle tc_0, \text{pass}, 5 \rangle$ if it takes 5 time units to execute tr_0 .

To access the time information we define the **performance verdict function for a single test run** $tr = \langle tc, v, t \rangle$ as follows:

$$\mathcal{T}(tr) = t. \quad (8)$$

Example 10 For tr_0 from the previous example, we obtain $\mathcal{T}(tr_0) = 5$.

To combine **performance and behaviour** information we use the projection operator π to obtain time from a given test run $tr = \langle tc, v, t \rangle$:

$$\pi_{\text{pass}}(\mathcal{T}(tr)) = \begin{cases} t & \text{if } v = \text{pass} \\ \# & \text{otherwise} \end{cases}. \quad (9)$$

By analogy we define projections for **inconc** and **fail**.

Example 11 For tr_0 defined in Example 9 we obtain $\pi_{\text{pass}}(\mathcal{T}(tr_0)) = 5$, $\pi_{\text{inconc}}(\mathcal{T}(tr_0)) = \#$, $\pi_{\text{fail}}(\mathcal{T}(tr_0)) = \#$.

4.2 Performance verdict functions for a sequence of test runs

Performance verdict functions for a sequence of test runs can use arithmetical functions, such as *sum*, *average*, *max*, *min*, and *count* to express restrictions over test execution time:

$$\bigotimes \mathcal{T}(TR) = \bigotimes_{tr \in TR} \mathcal{T}(tr), \quad (10)$$

where \bigotimes is a place holder for any arithmetical operation.

Example 12 Adding performance information to TR_0 from Example 4 we get $TR_0 = \langle tr_0, tr_1, tr_2 \rangle = \langle \langle tc_0, \text{pass}, 5 \rangle, \langle tc_1, \text{pass}, 10 \rangle, \langle tc_2, \text{inconc}, \perp \rangle \rangle$. If we assume that the maximum time for a single test run must not exceed value of 10, i.e., $\min \mathcal{T}(TR) \leq 10$, then TR_0 does not satisfy this condition, as tr_2 was executed in undefined time.

From the example above we can see that such restrictions over test execution times are not expressive enough to cover all practical situations. To have a more useful mechanism we need again to combine information on **behaviour and performance** using the projection again:

$$\bigotimes \pi_{\text{pass}}(\mathcal{T}(TR)) = \bigotimes_{tr \in TR} \pi_{\text{pass}}(\mathcal{T}(tr)). \quad (11)$$

Example 13 With the aforementioned extension we are able to express the following arbitration for TR_0 from Example 12 $\max \pi_{\text{pass}}(\mathcal{T}(TR)) \leq 5$ which means, we restrict maximal time for all test runs in TR with pass verdicts to 5 time units. For given TR_0 this is too restrictive, as tr_1 needs 10 time units to pass.

4.3 Arbitrations over performance verdict functions

Based on performance verdict functions it is possible to express test related to performance and accessibility of SUT.

Example 14 Combining performance verdict functions with logical operations we can express arbitrations with conjunction, such as one expressed by the following formula:

$$\max \pi_{\text{pass}}(\mathcal{T}(TR)) \leq 5 \wedge \max \pi_{\text{inconc}}(\mathcal{T}(TR)) < \perp.$$

The arbitration ensures that the maximal execution time of test runs that obtained a pass verdict is lower or equal to 5 time units and no time-out occurred for test runs that obtained an inconclusive verdict.

5 Generic Arbitrations and Test Reports

Based on definitions introduced in Sections 3 and 4 we can define generic arbitrations (Section 5.1). To achieve more flexibility in arbitration strategies we introduce additional labelling (Section 5.2) and for better overview we integrate arbitrations into test reports (Section 5.3).

5.1 Arbitrations over behavioural and performance verdict functions

With all functions introduced in the previous Sections 3 and 4 it is possible to cover a broad range of expressions that are important for practical applications. It is possible to combine behavioural and performance verdict functions to formulate arbitration strategies which cover both aspects.

Example 15 We can combine behavioural and performance aspects by logical conjunction of previously defined arbitrations from Example 7 and Example 14.

5.2 Arbitration Labelling

Additionally we introduce arbitration labelling to make the arbitration mechanism more generic and flexible. To each arbitration we can assign a set of labels. We consider labels related to development stages of SUT, different parts of the SUT and labels indicating verdict types. With labels fine tuning of restrictions over test runs depending on a project progress are possible. Analysis of border cases enables to specify the focus of future development of the system.

Example 16 Consider the real-time database example described in Section 1.1. We can consider a weak arbitration associated with a first system release label. The arbitration can state that the system may take up to 10 seconds for 2% of all searches. In the ideal situation the system should always return the complete list of all matching records within one second, but in a real situation due to the huge amount of data, this is not always possible. In the development phase it might be interesting to investigate cases with a part of the resulting list obtained within the predefined time or cases with the complete result list where the searching process took a little longer than one second.

5.3 Test Reports

To utilise all information gathered during test execution by the help of the verdict functions described in Sections 3 and 4 test reporting is of great significance. A test report manager essentially takes care of two jobs (Fig. 2): firstly, it has to evaluate the information gathered by former test runs, and secondly, it has to present the evaluated data by generating a test report. In order to evaluate the test run information, the report manager queries the database and receives the logging data. On the basis of this, it evaluates the verdict functions and generates a user configurable test report.

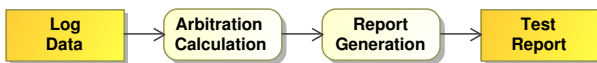


Figure 2. From log data to a test report

Based on the experience from software system development in the telecommunication and real-time databases domains we gathered the following **requirements for the report manager**:

- *flexibility of report definitions* — users should have the possibility to select the required level of detailedness (how the test runs should be selected and grouped), types of used arbitrations (selection of expressions and labels), and the type of visualisation (as for example in BIRT),
- *traceability to artefacts* involved in the testing process — the test report should enable navigation to the artefacts, such as, tested parts of SUT in appropriate versions, test specifications and test data used at the point

of testing, and detailed log information for single test runs,

- *progress report* support — reports over regression tests showing evolution of SUT and progress statistics, e.g. as defined in [5].

6 Conclusion and future work

The lack of verdicts combining performance measurement and behaviour in other testing tools engaged the Telling TestStories project. With the arbitration techniques introduced in this paper and the well stated but flexible reporting the aforementioned goals can be achieved, i.e., better understanding of testing process, monitoring of implementation progress, estimation of overall software system reliability and reduction in the development cost. To the best of our knowledge no other framework has the ability to deal with both aspects and enables arbitration in combination with reporting.

In the future we will integrate the presented concepts into the Telling TestStories framework with implementation of verdict functions and arbitrations as OCL expressions and implementation of reports in BIRT. After the implementation steps evaluation on a case study from the telecommunication area is planned.

7 Acknowledgements

The research herein is partially conducted within the competence network Softnet Austria (www.soft-net.at) funded by the Austrian Federal Ministry of Economics (bm:wa), the province of Styria, the Steirische Wirtschaftsförderungsgesellschaft mbH. (SFG), and the city of Vienna in terms of the center for innovation and technology (ZIT). Moreover a part of the research is conducted within Telling TestStories project funded by TransIT (www.transit.ac.at).

References

- [1] R. Breu, J. Chimiak-Opoka, and C. Lenz. A novel approach to model-based acceptance testing. In *Proceedings of the 4th MoDeVVA workshop*, pages 13–22, 2007.
- [2] C. Willcock et al. *An Introduction to TTCN-3*. John Wiley and Sons, 2005.
- [3] P. Baker et al. *Model-Driven Testing — Using the UML Testing Profile*. Springer-Verlag, Berlin Heidelberg, October 2007.
- [4] R. Hierons. Verdict functions in testing with a fault domain or test hypotheses. *ACM Trans. Softw. Eng. Methodol.*, 14(2):246–246, 2008. to appear.
- [5] D. Freeman. Practical test reporting. on-line publication at <http://www.stickyminds.com/>, 2008. accessed on 2008–09–05.