A Human-Centred Framework for Supporting Agile Model-Based Testing

Maria Spichkova[1], Anna Zamansky[2]

[1] RMIT University, Australia, maria.spichkova@rmit.edu.au
[2] University of Haifa, Israel, annazam@is.haifa.ac.il

Abstract. The successful application of model-based testing (MBT) heavily relies on constructing a complete and coherent *model* of a system. This implies that inconsistency, incompleteness, or inaccuracy due to human error bear significant consequences. We propose a formal framework for MBT which we call AHR: *agile, human-centred and refinement-oriented*. AHR captures an iterative construction of models and test plans, as well as supports refinements at different levels of abstraction.

1 Introduction

Model-based testing (MBT) is a technique for generating test cases from system model. Testers using this approach concentrate on a data model and generation infrastructure instead of hand-crafting individual tests, cf. [2, 4]. MBT heavily relies on *models* of a system and its environment to derive test cases for the system [19]. A system model is a result of the process of *abstraction*, the aim of which is a simplification of the complexity of a system and its environment. If the system is complex enough, however, several refinement steps may be required, each time using a more detailed representation of the system. Testing methodologies for complex systems therefore often integrate different abstraction levels of the system representation. The crucial points for each abstraction level are (*i*) whether we really require the whole representation of a system to analyse its core properties, and (*ii*) which test cases are required on this level [11]. As pointed out in [8], MBT makes sense only if the model is more abstract than the system under test. This implies that only behaviour encoded in the model can be tested, and that different levels of abstraction must be bridged. Modelling in MBT remains a strictly human activity, and the successful employment of MBT techniques heavily relies on the human factor. [4] mentions the steep learning curve for modelling notations as one barrier in the adoption of MBT in industry. Another barrier is the lack of state-of-the-art authoring environments, which can provide (semi-automatic) support for the human tester and help minimise the number of human errors as well as their impact.

Construction of complex models heavily relies on tacit human knowledge and therefore will always remain the task of a human tester, escaping full automatisation. The complexity and error-prone nature of this task, however, calls for

Copyright © by the paper's authors. Copying permitted only for private and academic purposes.

In: S. España, M. Ivanović, M. Savić (eds.): Proceedings of the CAiSE'16 Forum at the 28th International Conference on Advanced Information Systems Engineering, Ljubljana, Slovenia, 13-17.6.2016, published at http://ceur-ws.org

CEUR-WS.org/Vol-1612/paper14.pdf

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

more emphasis on human-centred approaches in automatic support of model-based testing. There are numerous works on human error in software development [6, 15] and human-oriented software development [12, 13, 17].

In this paper we introduce AHR (agile, human-oriented, refinement-oriented), a formal framework for integrating human-centred considerations into MBT with multiple levels of abstraction. This framework extends the ideas of *Human-Centred Agile Test Design* (HCATD), cf. [20], where it is explicitly acknowledged that the tester's activity is not error-proof: human errors can happen, both in the model and the test plan, and should be taken into account. HCATD combines agile modelling with test planning, with the idea to explicitly make room for inconsistency, incompleteness and inaccuracy of models and test plans in MBT. The discovery of an error or incomplete information may cause the tester to return to the model and refine it, which in its turn may induce further changes in the existing test plan. Agile software development process focuses on facilitating early and fast production of working code [9] by supporting iterative, incremental development. The term "agile" in AHR is meant to reflect the iterative and incremental nature of the process of modelling and test planning. We demonstrate the applicability of AHR using an example from the domain of combinatorial testing of cyber-physical systems. The framework can be seen as the first step towards developing tools and environments supporting the human modeller/tester in agile MBT.

2 The AHR Framework

The proposed AHR framework has three core features:

- **A** *Agile:* Both the test plan and system model at each abstraction level of modelling/implementation are agile in the sense that they are (only) sufficiently complete, accurate and consistent (with coverage requirements); error correction and specification extension are iteratively performed.
- **H** *Human-centred:* Error correction and specification extension/completion can be supported by posing series of queries or issuing alerts to the tester.
- **R** *Refinement-oriented:* Refinement is performed in the framework in several ways: (i) the usual static refinement of system properties when moving between abstraction levels, (ii) dynamic refinement of system properties and test plans as a result of information completion or error correction.

Let S be the system under test. Assume that we construct a model with m levels of abstraction. We say that S can be completely described at each level l by the set $\mathbb{PROP}^l(S)$ of its properties. The main two tasks of the human tester in the process of MBT are (1) to construct an appropriate abstraction M of S, and (2) to propose a test suite that validates S against M according to some chosen coverage requirements. For performing the task (1), we have to decide which properties of the system are important to model/implement (and, respectively, to test) and which need to be abstracted away. Thus, we have to partition the set $\mathbb{PROP}l(S)$ into two disjoint subsets: $\mathbb{LPROP}^l(S)$ – the properties

to be modelled/implemented and tested, and $\mathbb{ABSTR}^l(S)$ – the properties to be abstracted away, knowingly or unknowingly.

$$\mathbb{LPROP}^l(S) \cup \mathbb{ABSTR}^l(S) = \mathbb{PROP}(S)$$
$$\mathbb{LPROP}^l(S) \cap \mathbb{ABSTR}^l(S) = \emptyset$$

The properties $\mathbb{LPROP}^l(S)$ might include the pre- and post-conditions of actions. We denote the pre- and post-conditions of an action Act on the level l by $Pre^l(Act)$ and $Post^l(Act)$ respectively.

On each abstraction level the traceability between the system properties and the corresponding tests is crucial for our approach (cf. also Figure 1). If the information is not important on the current level, it could influence on the overall modelling result after some refinement steps, i.e., at more concrete levels that are closer to the real system in the physical world. Therefore, while specifying system we should make all the decisions on abstraction in the model transparent and track them explicitly. In the case of contradiction between the model and the real system this would allow us to find the problem easier and faster.

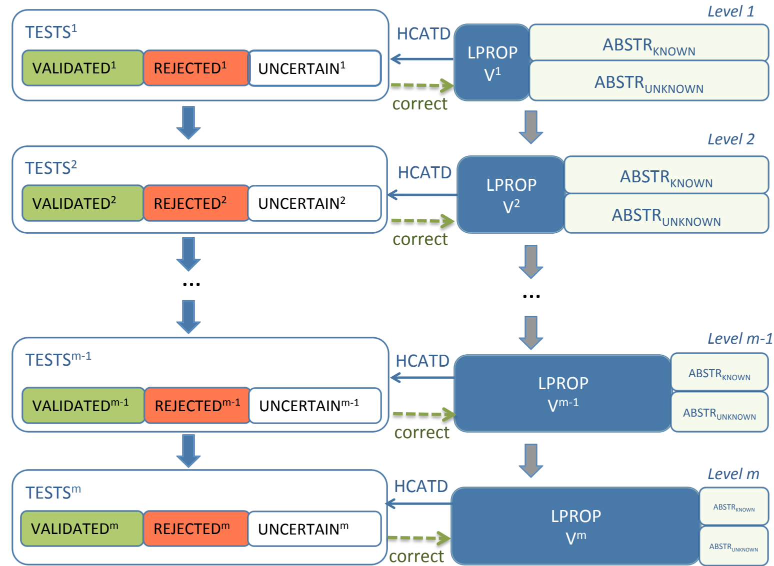


Fig. 1. AHR Framework

To introduce an explicit representation of incompleteness, which is manageable and traceable at different levels of abstraction, we suggest to divide the set $\mathbb{ABSTR}^l(S)$ into two disjoint subsets, $\mathbb{ABSTR}_{\mathbb{KNOW}}{}^l(S)$ and $\mathbb{ABSTR}_{\mathbb{UNKNOW}}{}^l(S)$:

$$\mathbb{ABSTR}_{\mathbb{KNOW}}{}^l(S) \cup \mathbb{ABSTR}_{\mathbb{UNKNOW}}{}^l(S) = \mathbb{ABSTR}^l(S)$$
$$\mathbb{ABSTR}_{\mathbb{KNOW}}{}^l(S) \cap \mathbb{ABSTR}_{\mathbb{UNKNOW}}{}^l(S) = \emptyset$$

This allows us to separate the properties of the system from which we abstract intentionally from those, from which we abstract a unknowingly. The $\mathbb{ABSTR}_{\mathbb{UNKNOW}}{}^{l}(S)$ properties are not identified/classified at level l due to lack of information (which may be due to error/omission). Thus, on each level l we generally operate with *three* sets of properties:

- $\mathbb{LPROP}^{l}(S)$ – properties the tester decided to include at this level,
- $\mathbb{ABSTR}_{\mathbb{KNOW}}{}^{l}(S)$ – properties from which he knowingly abstracts at l;
- $\mathbb{ABSTR}_{\mathbb{UNKNOW}}{}^{l}(S)$ – properties which he unknowingly abstracts at l.

With each refinement step the tester moves some part of system's properties from the set $\mathbb{ABSTR}$ to the set $\mathbb{LPROP}$. We can say that in some sense the set $\mathbb{ABSTR}$ represent the termination function for the modelling process.

In a similar manner, the set of all tests is divided into three mutually disjoint subsets on each level l:

- $validated^{l}$: the tester confirmed these tests as executable on this abstraction level according to some chosen confirmation strategy;
- $rejected^{l}$: the tester rejected these tests as impossible or irrelevant on this abstraction level;
- $uncertain^{l}$: the tester has not classified these tests to be validated/rejected, as not enough information has been provided for the classification.

3 Combinatorial Test Design with AHR

A type of MBT which uses particularly simple models is combinatorial test design (CTD), cf. [3, 5, 10]. In CTD a system is modelled using a finite set of system parameters $\mathbf{A} = \{\mathbf{A}_1, \dots, \mathbf{A}_n\}$. Each of the parameters is associated with a set of corresponding values $\mathbf{V} = \{\mathbf{V}(\mathbf{A_1}), \dots, \mathbf{V}(\mathbf{A_n})\}$. The main challenge of CDT is to optimise the number of test cases, while ensuring the coverage of given conditions. Tai and Lei [18] have shown in their experimental work that a test set covering all possible pairs of parameter values can typically detect 50-75% of the bugs in a program. In what follows we demonstrate an application of the AHR framework for CDT. In our approach, we suggest to analyse *interactions* between the different values of the parameters, i.e., elements of the form $\mathcal{I} \subseteq \bigcup_1^n \mathbf{V}(\mathbf{A}_i)$, where at most one value of each parameter may appear. An interaction of size n (where some value of each system parameter appears) is a *scenario* (or *test*). We say that a set of scenarios T *covers* a set of interactions C if for every $c \in C$ there is some $t \in T$, such that $c \subseteq t$. A *combinatorial model* $\mathcal{E}$ of the system is a set of scenarios, which defines all tests executable in the system. A *test plan* is a triple $Plan = (\mathcal{E}, C, T)$, where $\mathcal{E}$ is a combinatorial model, C is a set of interactions called coverage requirements, and T is a set of scenarios called tests, where T covers C.

One of the most standard coverage requirements is *pairwise testing* [7, 18]: considering every (executable) pair of possible values of system parameters. In the above terms, a pairwise test plan can be formulated as any pair of the form

$Plan = (\mathcal{E}, C_{pair}(\mathcal{E}), T)$, where C_{pair} is the set of all interactions of size 2 which can be extended to scenarios from $\mathcal{E}$.

Typically, the CTD methodology is applied in the following stages. First the tester constructs a combinatorial model of the system by providing a set scenarios which are executable in the system. After choosing the coverage requirements, the second stage is constructing a test plan, i.e., proposing a set of tests over the model, so that full coverage with respect to a chosen coverage strategy is achieved. The set of parameters $\mathbf{A}$ as well as the sets of corresponding values $\mathbf{V}$ will be refined and extended along with the refinement and extension of the sets of properties. Respectively, the sets of tests have to be also refined. When a new level $l+1$ is created by specifying $\mathbb{LPROP}^{l+1}$ and $\mathbf{V}^{l+1}$, and some parameters and their values are unchanged while refining the system from level l to level $l+1$, then a number of corresponding tests are still unchanged too, which means that we can also reuse their marking as validated/rejected/uncertain. If we trace the refinement relations not only between the properties but also between test plans, this might help to correct possible mistakes more efficiently, as well as provide additional support if the system model is modified. In AHR, we use upper indices on the test names to denote that the test belongs to a particular abstraction level, e.g., $test_5^3$ would denote that the $test_5$ defined on the *Level 3*.

4 Example Scenario: A Cyber-Physical System

An important domain in which modelling with different levels of abstraction is particularly beneficiary is *cyber-physical systems* (CPS). Our early work on specification of CPS on abstract level was presented in [14].

Let us consider a cyber-physical system system with two robots R_1 and R_2 interacting with each other. Using the AHR framework, we model each robot on *Level 1* by two parameters, GM and P:

- GM represents the mode of the robot's grippers, which can be either either closed (to hold an object) or open. We specify two system parameters of this type on *Level 1*, GM_1 and GM_2 specifying the gripper modes of R_1 and R_2 respectively, where $\mathbf{V}(GM_1) = \mathbf{V}(GM_2) = \{open, closed\}$.
- P represents the robot's position. On *Level 1*, we assume to have only three possible positions for each robot. In the system model on *Level 1*, we have two system parameters of this type, P_1 and P_2 specifying the positions of R_1 and R_2 respectively, where $\mathbf{V}(P_1) = \mathbf{V}(P_2) = \{pos_1, pos_2, pos_3\}$.

In what follows let us assume *pairwise coverage requirements*. We specify two meta-operations *Give* and $Take$ to model the scenario when one robot hands an object to another robot. A meta-operation *Give* in which R_1 gives an object to R_2 can only be performed when the gripper of R_1 is closed, the gripper of R_2 is open, and the grippers of both robots are in the same position. This means that not all test cases are executable and further restrictions should be imposed. Suppose, however, that the information on the position is erroneously omitted, because of a human error on specification level while defining the set of $\mathbb{LPROP}^1$. Thus,

$\mathbb{LPROP}^1 = \{Pre^1(Give)\}$, where $Pre^1(Give) = \{GM_1 = closed, GM_2 = open\}$. This induces a system model with the corresponding tests (cf. Table 1).

Table 1. Example scenario: System model on *Level 1*

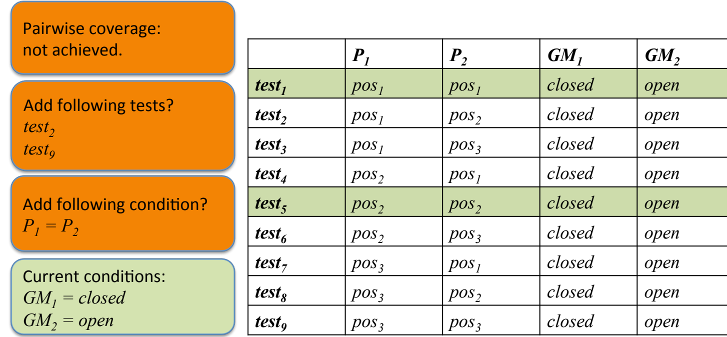
	P_1	P_2	GM_1	GM_2
$test_1$	pos_1	pos_1	closed	open
$test_2$	pos_1	pos_2	closed	open
$test_3$	pos_1	pos_3	closed	open
$test_4$	pos_2	pos_1	closed	open
$test_5$	pos_2	pos_2	closed	open
$test_6$	pos_2	pos_3	closed	open
$test_7$	pos_3	pos_1	closed	open
$test_8$	pos_3	pos_2	closed	open
$test_9$	pos_3	pos_3	closed	open

Initially, all tests are be marked as *uncertain*. The tester then goes on to construct a test plan by selecting two tests from the table, for example, $test_1$ and $test_5$. In this test plan, the tester erroneously omitted a test case including pos_3. Once the tester submits the test plan, $test_1$ and $test_5$ are marked as *validated*. At this point the tester's mistake may be discovered, as pairwise coverage is not achieved: e.g., the interactions $\{P_1 : pos_1, P_2 : pos_2\}$ and $\{P_1 : pos_3, P_2 : pos_3\}$ remain uncovered. This can be either due to the fact that the tester considered non-executable tests as possible or forgot to add some tests.

A human-oriented solution to this kind of problems would be issuing a query to prompt the tester to either extend the logical condition with $P_1 = P_2$ (thus removing the interaction $\{P_1 : pos_1, P_2 : pos_2\}$ from coverage requirements) or extend the test plan with $test_9$. To provide a better overview of the plan conditions while analysing whether a new conditions and/or tests should be added, we also suggest to provide an option *show the current conditions*. When the tester decides to mark $test_9$ as *validated* and to add the logical condition $P_1 = P_2$ to the set $\mathbb{LPROP}^1$, the framework will update the set $\mathbb{LPROP}^1$ and notify the tester that the pairwise coverage is achieved under the current selected conditions. After the corresponding corrections, $\mathbb{LPROP}^1 = \{Pre^1(Give)\}$, where

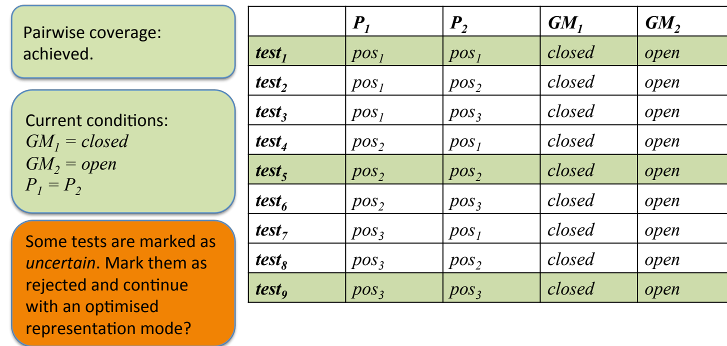
$$Pre^1(Give) = \{GM_1 = closed, GM_2 = open, P_1 = P_2\}$$

As the next step, the framework reminds the tester that a number of tests are still marked as *uncertain* (cf. Figure 3). The tester might either accept with the current marking or mark all the uncertainties as *rejected* to switch to an optimised view, where only *validated* tests are presented to increase the readability.



	P_1	P_2	GM_1	GM_2
$test_1$	pos_1	pos_1	closed	open
$test_2$	pos_1	pos_2	closed	open
$test_3$	pos_1	pos_3	closed	open
$test_4$	pos_2	pos_1	closed	open
$test_5$	pos_2	pos_2	closed	open
$test_6$	pos_2	pos_3	closed	open
$test_7$	pos_3	pos_1	closed	open
$test_8$	pos_3	pos_2	closed	open
$test_9$	pos_3	pos_3	closed	open

Fig. 2. Example scenario (*Level 1*): Pairwise coverage is not achieved



	P_1	P_2	GM_1	GM_2
$test_1$	pos_1	pos_1	closed	open
$test_2$	pos_1	pos_2	closed	open
$test_3$	pos_1	pos_3	closed	open
$test_4$	pos_2	pos_1	closed	open
$test_5$	pos_2	pos_2	closed	open
$test_6$	pos_2	pos_3	closed	open
$test_7$	pos_3	pos_1	closed	open
$test_8$	pos_3	pos_2	closed	open
$test_9$	pos_3	pos_3	closed	open

Fig. 3. Example scenario (*Level 1*): Pairwise coverage is achieved

5 Conclusions and Future Work

This paper[3] introduces the AHR framework for supporting an agile MBT. The framework explicitly acknowledges that the human tester as well as the system designer may make mistakes that need to be corrected. Error correction and specification extension/completion is supported by posing series of queries or issuing alerts to the tester. AHR involves several abstraction levels of modelling and testing, where models and test planes can be iteratively updated, completed and improved during the development process. To illustrate the core features of the framework, we have presented an example scenario from the domain of cyber physical systems. AHR has the potential to facilitate the adoption of MBT in industry, increasing the efficiency of the human tester and providing opportunities for collaborative efforts. To increase the productivity of our approach, we might embed it into facilities as the Virtual Experiences Laboratory [1, 16], where the interoperability simulation and testing are performed remotely.

[3] The second author was supported by The Israel Science Foundation under grant agreement no. 817/15.

References

1. J.O. Blech, M. Spichkova, I. Peake, and H. Schmidt. Cyber-virtual systems: Simulation, validation & visualization. In *International Conference on Evaluation of Novel Approaches to Software Engineering*, 2014.
2. S.R. Dalal, A. Jain, N. Karunanithi, J.M. Leaton, C.M. Lott, G.C. Patton, and B.M. Horowitz. Model-based testing in practice. In *International Conference on Software Engineering*, pages 285–294. ACM, 1999.
3. E. Farchi, I. Segall, R. Tzoref-Brill, and A. Zlotnick. Combinatorial testing with order requirements. In *International Conference on Software Testing, Verification and Validation Workshops*, pages 118–127. IEEE, 2014.
4. W. Grieskamp. Multi-paradigmatic model-based testing. In *Formal Approaches to Software Testing and Runtime Verification*, pages 1–19. Springer, 2006.
5. R. Kuhn, R. Kacker, Y. Lei, and J. Hunter. Combinatorial software testing. *IEEE Computer*, 42(8):94–96, 2011.
6. T. Mioch, J.-P. Osterloh, and D. Javaux. Selecting human error types for cognitive modelling and simulation. In *Human modelling in assisted transportation*, pages 129–138. Springer, 2011.
7. C. Nie and H. Leung. A survey of combinatorial testing. *ACM Comput. Surv.*, 43(2):11:1–11:29, February 2011.
8. A. Pretschner. Model-based testing in practice. In *FM 2005: Formal Methods*, pages 537–541. Springer, 2005.
9. B. Rumpe. Agile test-based modeling. In *International Conference on Software Engineering Research & Practice*. CSREA Press, 2006.
10. I. Segall, R. Tzoref-Brill, and A. Zlotnick. Common patterns in combinatorial models. In *International Conference on Software Testing, Verification and Validation (ICST)*, pages 624–629. IEEE, 2012.
11. M. Spichkova. Architecture: Requirements + Decomposition + Refinement. *Softwaretechnik-Trends*, 31:4, 2011.
12. M. Spichkova. Human Factors of Formal Methods. In *IADIS Interfaces and Human Computer Interaction 2012*. IHCI 2012, 2012.
13. M. Spichkova. Design of formal languages and interfaces: formal does not mean unreadable. In *Emerging Research and Trends in Interactivity and the Human-Computer Interface*. IGI Global, 2013.
14. M. Spichkova and A. Campetelli. Towards system development methodologies: From software to cyber-physical domain. In *Formal Techniques for Safety-Critical Systems*, 2012.
15. M. Spichkova, H. Liu, M. Laali, and H. Schmidt. Human factors in software reliability engineering. *Workshop on Applications of Human Error Research to Improve Software Engineering*, 2015.
16. M. Spichkova, H. Schmidt, and I. Peake. From abstract modelling to remote cyber-physical integration/interoperability testing. In *Improving Systems and Software Engineering Conference*, 2013.
17. M. Spichkova, X. Zhu, and D. Mou. Do we really need to write documentation for a system? In *Model-Driven Engineering and Software Development*, 2013.
18. K.-C. Tai and Y. Lei. A test generation strategy for pairwise testing. *IEEE Transactions on Software Engineering*, 28(1):109–111, 2002.
19. M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, 2012.
20. A. Zamansky and E. Farchi. Helping the tester get it right: Towards supporting agile combinatorial test design. In *Human-Oriented Formal Methods*, 2015.