Athens Journal of Sciences- Volume 6, Issue 4 – Pages 265-286

Testing Artificial Intelligence by Customers' Needs

By Thomas Fehlmann[*] *& Eberhard Kranich*[‡]

Artificial Intelligence (AI) is everywhere, nowadays. No longer limited to computer laboratory, it now sets up media campaigns, influences people, decides elections, protects property, and drives cars. The principles of AI are quite old; most of them originate from the early times of computer science and had been discussed in the seventies and the eighties. But they were theoretical concepts as computer power was a scarce resource, and not enough data was available to feed the perception of that time. However, now AI governs even safety-criminal systems. How can it be tested? The answer is surprisingly simple: consider the system's goals.

Keywords: *Artificial Intelligence, Neural Networks, Safety Metrics, Software Metrics, Support Vector Machines, Test Coverage, Test Metrics.*

Introduction

The principles of AI are classification of entities and the solution of the equation $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{y}$, where $\boldsymbol{x}$ and $\boldsymbol{y}$ are vectors in spaces of different dimensions and semantics. For instance, $\boldsymbol{y}$ could be the observable behavior of people or extrasolar planets, and $\boldsymbol{x}$ the unknown cause of it. Traditional solution methods are regression, where both $\boldsymbol{x}$ and $\boldsymbol{y}$ are measurable, or eigenvector methods, where the $\boldsymbol{x}$, is not directly measurable but the correlation between the two is measurable. If there is not much known about the transfer function $\boldsymbol{f}$, neural networks can be set up that learn the transfer function, based on experience.

If a car uses an image recognition system, it must learn to distinguish between people walking, running, children playing, waiting, and bicycles riding, or being walked. The system should also be able to recognize people and things if not seen in full, even partly hidden behind a bush, by fog, or at night. Such a system is programmed to learn; neither image characteristics nor pattern recognition algorithms are programmed into it.

Unfortunately, such neural networks do not only learn, they also unlearn. Van Gerven et al. (2018) have shown that they can get distressed ("mad") as any neural information processing, similar to humans. Before I dare to sit into an autonomous car, I probably want to know whether and when this car with its current state of the learning system had passed its last test.

[*]Senior Researcher, Euro Project Office AG, Switzerland.
[‡]Senior Researcher, Euro Project Office, Germany.

https://doi.org/10.30958/ajs.6-4-3 doi=10.30958/ajs.6-4-3

Our Approach

This requires the ability to test software-intense systems autonomously. It must be possible to test a car's image analysis capabilities in regular intervals to see whether it still works as expected; thus, detecting "madness" early enough to avoid damage. Whether the old capabilities still work, it needs being tested.

Moreover, tests must evolve as well. They cannot be static; test suites need to expand for new learnings, new environments, and new standards and regulations.

This is *Autonomous Real-time Testing* (ART). ART needs AI techniques to generate enough test cases, and thanks to this expandability ART is capable of testing other AI.

Why Learning is Not Good Enough without Testing

The death of Elaine Herzberg (August 2, 1968 - March 18, 2018) was the first recorded case of a pedestrian fatality involving an autonomous car, following a collision that occurred at around 10pm Mountain Standard Time (UTC-7) in the evening of Sunday, March 18, 2018 (The National Transportation Safety Board 2018). The following narrative is extracted from the said source.

Herzberg was pushing a bicycle across a four-lane road in Tempe, Arizona, United States, when she was struck by Volvo XC90 taxi outfitted with a sensor system, operated under test conditions by Uber. Since 2015, Uber conducted tests with various levels of automation in Arizona. The car was operating in self-drive mode with a human safety backup driver sitting in the driving seat. Following the collision, Herzberg was taken to the hospital where she died of her injuries.

According Uber, the accident was largely caused by the software that decides how the car should react to objects it detects. The car's sensors detected the pedestrian, who was crossing the street with a bicycle. Uber's software first registered Elaine Herzberg on lidar six seconds before the crash — at the speed it was traveling, that puts first contact at about 115m away. As the vehicle and pedestrian paths converged, the self-driving system software classified the pedestrian first as an unknown object, then as a vehicle, and then as a bicycle with varying expectations of future travel path. The software decided it did not need to react right away. Like other autonomous vehicle systems, Uber's software can ignore "false positives," or objects in its path that are not an obstacle for the vehicle, such as a plastic bag floating over a road.

Then, 1.3 seconds before impact, which is to say about 24m away, the self-driving system determined that an emergency braking maneuver was needed to mitigate a collision. According to Uber, emergency braking maneuvers are not enabled while the vehicle is under computer control, to reduce the potential for erratic vehicle behavior. The vehicle operator is relied on to intervene and act. The system is not designed to alert the operator. The Volvo model's built-in safety systems — collision avoidance and emergency braking, among other things — were also disabled while in autonomous testing mode.

The self-driving system data showed that the vehicle operator intervened less than a second before impact by engaging the steering wheel. The vehicle speed at impact was 62km/h. The operator began braking less than a second after the impact. The data also showed that all aspects of the self-driving system were operating normally at the time of the crash, and that there were no faults or diagnostic messages.

The dead of Elaine Herzberg raises one major question: Why were the visual recognition systems tested in real life situations only, instead under labor conditions before hitting the road?

A Primer on Metrics for Testing Software

A *Test* is a finite collection of *Test Stories.* Test stories are finite collections of *Test Cases*, characterized by some common business value delivered. Business value is defined by *User Stories*. Test stories are related to user stories but typically not the same. Test stories address more than just one user story, combining different aspects from loosely related sources.

Test cases $\{x_1, x_2, \ldots, x_n\} \rightarrow y$ start with a set of preconditions $x_1, x_2, \ldots, x_n$ and yield some known response y. Test cases always contain weakest assertions; thus, inequalities or range specification rather than sample numbers; see the book of the author (Fehlmann 2016, p. 319ff).

A Model for Software Functionality: ISO/IEC 19761 COSMIC

When consulting the ISO/IEC/IEEE 29119 testing standard (ISO/IEC/IEEE 29119-4, 2015), it astonishes that part 4 identifies 23 different so-called *Test Coverage Items*. But tests primarily address software functionality. It is unnecessary to define extra “Items” that undergo testing.

Functional models are available and in use since the past 40 years for sizing software. Tests cover its model elements. We chose the ISO/IEC 19761 COSMIC standard (ISO/IEC 19761 2011). This model of software functionality consists of *Data Movements*, moving *Data Groups* from one *Object of Interest* into another. Thus, there exists only one test coverage item: obviously, the data movement.

Data Movement Maps

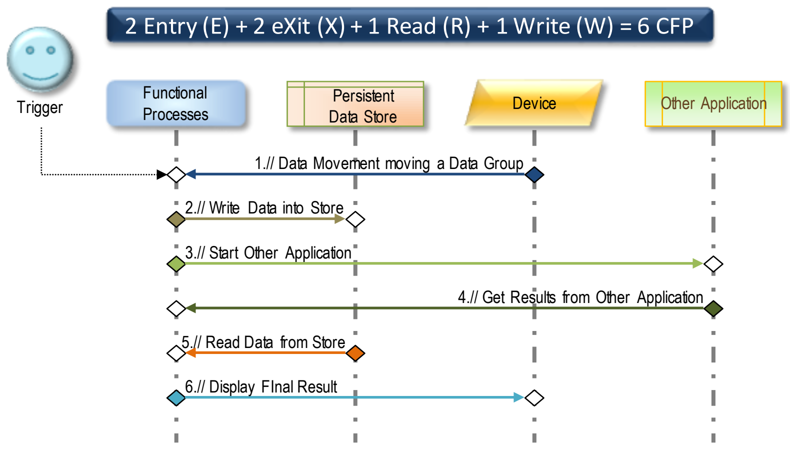
A piece of software connecting objects of interest that represent functionality, persistent stores, devices and other applications, can be modeled as *Data Movement Maps*. The connectors are called *Data Movements*.

Data movement maps have some resemblance to *UML Sequence Diagrams* (Bell 2004) but with less detail, and sequencing is not prescribed. The advantage of this representation is that size is immediately visible; useful as size count according ISO/IEC 19761. Data movements always move a *Data Group*, which can be thought as a data record. A data movement counts for size only if its data group is unique.

Uniqueness is indicated by color-filled trapezes. Another move of same data group between the same objects within a COSMIC functional process lets the trapeze blank.

There are four kinds of data movements: An **E**ntry to some functional process; an e**X**it to some device or other application; **R**eading from and **W**riting into a permanent store. Counting data movements yields the *Functional Size* (Figure 1).

Figure 1. *Sample Data Movement Map*


2 Entry (E) + 2 eXit (X) + 1 Read (R) + 1 Write (W) = 6 CFP
Trigger
Functional Processes
Persistent Data Store
Device
Other Application
1.// Data Movement moving a Data Group
2.// Write Data into Store
3.// Start Other Application
4.// Get Results from Other Application
5.// Read Data from Store
6.// Display FInal Result

When executing a test case, it is straightforward to identify the data movements that are executed. The initial data movements are those whose data group meets the assertions made for test data; the last data movement meets the response assertion. The test case simply is represented by a sub-map of the data movement map for the app being tested.

Moreover, objects of interest can be expected to provide test stubs; this means that such objects can provide test data without executing all the data creation functionality that under normal operational conditions is needed. If there is some hardware in the loop, test stubs are needed anyway to simulate the sensors' or actuators' data supplied into the test.

Test Size

Test Size thus is the number of data movements needed to execute some test case for producing the test response. According COSMIC rules, moving the same data group is counted only once for size. However, since a test story consists of many test cases, a specific data movement is executing many times within a test story, typically with different test data. All test cases within a test story must be different from each other.

Test Intensity in turn is an average number characterizing how many times on average a data movement becomes part of test case. Since high test intensity, does not rule out that not all data movements are executed at least once in a test, *Test Coverage* remains an important indicator, specifying the percentage of data movements not covered with one test case in some test story.

The total size of a test story is the sum of all test case sizes executed within a test story, thus increasing test size when executing more test cases.

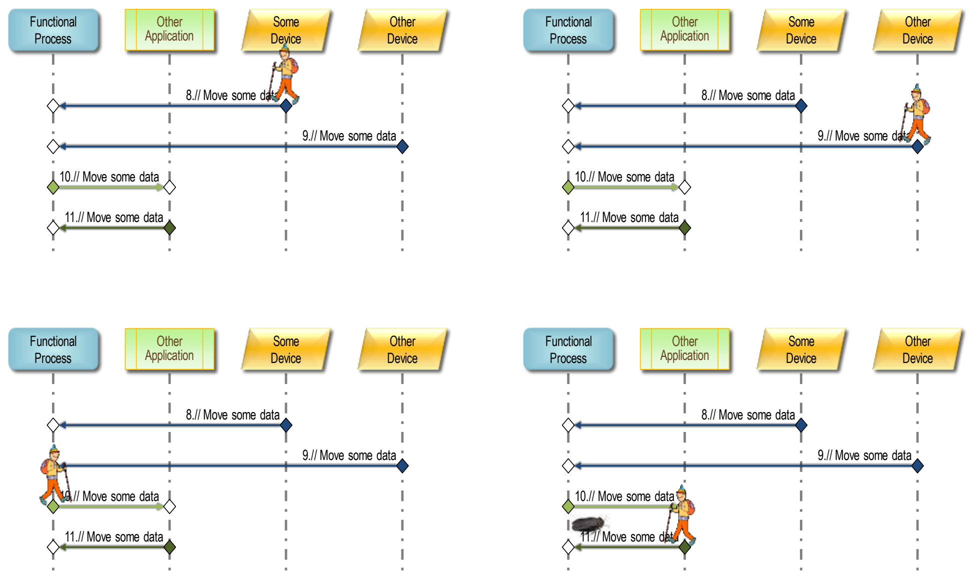
In statistics, test distribution indicates the degree to which test intensity differs within one test story, or within the full test suite. For practical purposes, such a metric seems not very telling, since it does not replace test coverage. It is rather expected that high business value increases test intensity while data movements moving irrelevant data are well tested with a few test cases only. Thus, test intensity depends from business value and is not normally distributed. Therefore, test distribution is not a meaningful indicator.

Test Walk

The data movement maps can be used to visualize tests cases. You can walk the tests, similar, but less in detail, as walking through code. Such visualization might help in crowd testing for identifying bugs found. The tester sees selected sequences in the data movement map; he can "walk" the data movements when planning or executing tests. This makes functionality visible to the development team, localizes defects that impact functionality, and supports communication between testers, users, and developers. Figure 2 shows how *Data Walker* walks four data movements of a test case and detects a bug at the fourth data movement.

A *Bug* is defined the traditional way: a test case that returns an unexpected response. Because our data walker can detect only one bug at a time, we are able to count defects unambiguously and thus define what defect density is. The maximum number of defects per test case is its test size. One test story only counts for a maximum of one defect per data movement

Figure 2. *Test Walk on Data Movement Maps (One Bug Found in Fourth Step)*



Defect Density

What is a defect? A defect means that the response does not meet the assertion of the response in the respective arrow term. It is therefore obvious that a defect relates to a test story. It refers to some data movement that exhibits the flawed response when executed by some specific test case.

Thus, counting defects become a limited task. You can count a maximum of one defect per data movement per test story. *Defect Density* is therefore a percentage of the total of defect opportunities. This definition opens the possibility to apply the usual Six Sigma techniques to defect density and defect distribution. Traditional defect counts obtained from counting the number of entries in a bug repository are not suitable for applying Six Sigma.

Test Coverage

Key among test metrics is *Test Coverage*. Test coverage has to do with users', or customers', values. It is useless to test pieces of software that deliver nothing visible to the user, or nothing that has any value. Test coverage refers to functionality, and not to code. Code implements functionality, and tests cover functionality, not code. Functionality can originate from anywhere, the cloud, other services. Code might provide other things that user functionality, e.g., additional security or traceability.

For defining test coverage, functionality needs evaluation in view of customer values. It is obvious that just counting whether any given piece of functionality is covered by tests does not yield useful metrics, because users see value in respective functionality differently.

Creating a Customer Needs Profile

The usual way of valuating functionality is by prioritizing user stories. Agile teams set priorities when selecting user stories for a sprint. Priorities are set by the product owner; however, the methods used for setting priorities are not standardized. Since product owner is the most difficult role in agile development, especially with Scrum (Schwaber and Beedle 2002), it is helpful to use a method dedicated to developing a product towards customer needs.

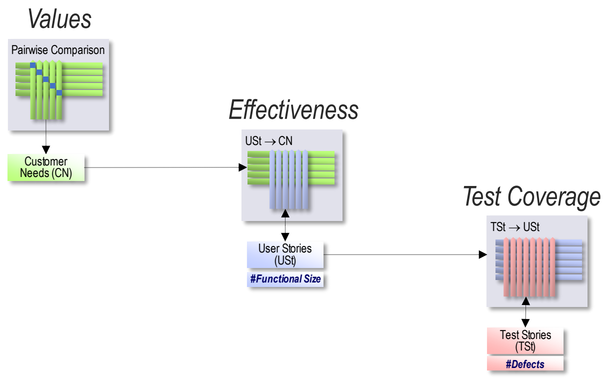
The method of choice is the *Analytical Hierarchy Process* (AHP), proposed by Saaty (2003) and used in Fehlmann (2016, p. 21), based on calculating Eigenvector solutions. The applicable ISO 16355 standard (ISO 16355-1:2015 2015) lists many more excellent alternatives for *Voice of the Customer* (Mazur 2014), e.g., *Net Promoter*® surveys (Fehlmann and Kranich 2014), and *Gemba* visits (Mazur and Bylund 2009). All these methods are part of *Quality Function Deployment* (QFD) (Fehlmann 2016, p. 16).

Effectiveness of the Implemented System

With customer needs established, user stories can easily be prioritized with a *Six Sigma Transfer Function* that maps user stories onto customers' needs. Transfer functions map causes to effects and have a *Convergence Gap* that indicates how well the effective profile of the transfer function matches the goal profile. The functional effectiveness transfer function uses the frequency of data movements needed for implementing the user stories. The resulting profile defines the user stories' priorities. These transfer functions are explained in Fehlmann (2016, p. 21ff, p. 196).

In turn, mapping test stories onto user stories, again using the frequency of data movements used in test cases, defines *Test Coverage*. The matrix looks familiar; tester uses it to assess coverage of code by tests. However, without proper test metrics you cannot assess a test coverage matrix for its quality. If the test cases in a series of test stories cover the user stories, and the transfer functions yields a satisfactory convergence gap, the tests cover customer needs exactly, up to the said convergence gap (Figure 3).

Figure 3. *Transfer Functions Overview*



Thus, the test coverage matrix represents a transfer function providing assurance that the test stories verify the correct implementation of the user stories. The convergence gap measures how well the correct implementation of user stories can be proved by these tests.

These tests do not prove anything else than the requirements expressed in the user stories. Adding user stories requires adding test stories. And as ever with transfer functions, there is no way of proving that the selected test stories are the only selection that works, not even the minimal one; you just can prove that the selected test stories work sufficiently well. Thus, the convergence gap, in turn, is a metric that can be used for test automation. It eliminates test stories that are not needed in view of the values of the customer.

Test Acceptance Criteria

Test coverage within a given convergence gap replaces traditional test completion criteria, as extensively spread out in the respective ISO standard (ISO/IEC/

IEEE 29119-4 2015, p. 125). Test cases are considered for inclusion in the test coverage transfer function only if passed when executing the test. The full test suite is passed if test coverage, computed with valid test cases only, results in a convergence gap below a certain limit. Currently, we believe a suitable limit is 0.1, corresponding to a 10% match between goal of testing and tests executed (Fehlmann 2016, p. 13ff).

How to Test Artificial Intelligence

Computer Vision and *Artificial Intelligence* (AI) overlap. AI is different from ordinary software by its capability to learn. This means, AI can adapt to new environments, data, images and videos. While AI can be used for other tasks, computer vision is concerned with the theory behind artificial systems that extract information from images. Areas of AI deal with autonomous planning or deliberation for robot systems to navigate through an environment. A detailed understanding of these environments is required to navigate through them. Information about the environment could be provided by a computer vision system, acting as a vision sensor and providing high-level information about the environment and the robot.

AI and computer vision share other topics such as pattern recognition and learning techniques. Consequently, computer vision is sometimes seen as a part of the AI field. Testing AI in computer vision obviously is not so straightforward; mainly, because it is not possible to predict what the correct outcome is. The test case might produce different responses, and all are correct at a given state of experience collection.

Recall that AI basically is sorting data into categories based on previous learning, or sample sets. In the Elaine Herzberg case, the Uber car did exactly that when its Lidar, and ten visual cameras, recognized the object moving towards the car's driveway. The difficulty was to find the right category. Humans encounter the same difficulty, when a biker enters the road from the pedestrian sidewalk. Expecting a pedestrian, they rapidly must adapt categories to a bicycle that follows different traffic rules than a pedestrian. Things become even more complicated if suddenly the pedestrian conjures up a skateboard, or a scooter. Traffic rules for the latter two conveyances are unknown, or do not exist. Humans are disturbed, and so are visual recognition systems, despite lidar and cameras.

Since the important contribution of the visual recognition system is categorization, it should be tested whether categories detected by the visual recognition system remain the same over its lifetime. But that is not enough. Behavior on certain sample image sequences should also remain stable – except if new learnings tell it otherwise. Obviously, tests must adapt to learnings. On the other hand, learning systems can become neurotically disturbed – mentally sick, like humans (Gerven and Bohte 2017). Thus, this is a case for *Autonomous Real-time Testing* (ART) (Fehlmann and Kranich 2017). Testing AI must be possible anytime, autonomous, without human intervention.

Baselining

You start testing AI as any other software:

- Identify the software under test
- Identify the goals of testing
- Draw a data movement map
- Calculate functional effectiveness
- Adjust scope of testing until goal and functional effectiveness converge
- Prepare the Test Stories:
  - Identify new test stories
  - Fill test stories by test cases
  - Calculate test coverage
- Repeat above three steps until test coverage converges

Perform the tests and validate test stories and test cases. Identify defects and remove them, or mitigate them, until your system is defect-free.

Extending Test Cases

Consider the AI domain when extending test cases. For instance, for traffic vehicles, use video sequences from traffic scenes already described in the test story. Use video sequences that have been used for deep learning and other who were not. You must manually classify the videos for the category of traffic it represents; it is therefore the same kind of work for testing as for learning.

With ART, you keep the test stories from the initial test suite stable while adding more test cases to improve test intensity and to detect more defects. For visual systems, the primary source for new test cases is new images and videos. Keeping test coverage good enough is somewhat easier than in other ART instances, since you only exchange test data. You do not change the aim of testing.

One primary source for new test cases is the contents of the data groups moved by the data movements in the ISO/IEC 19761 COSMIC model. Whatever can enter a certain functional process must become part of a test case, even for nothing else than proving it has no effect at all. Thus, according the combinatory logic approach, we combine all possible input as test data in the test cases and select those test cases that keep the convergence gap of the test coverage matrix small.

This is an automatic task; it has some resemblance to AI techniques as it means searching and categorizing data. The data stems from the data groups in the ISO/IEC 19761 COSMIC data movements; the combinatory algebra defines its structure. For more details see Fehlmann and Kranich (2018).

Interpreting Test Results

The aim of AI testing is to verify stable behavior in performing categorization as previously learned. This is different from human learning where humans should be able to interfere correct evaluations from their skills. As already mentioned, AI has not so much to do with the Latin origin of the word “Intelligence”, namely *intellegere,* read, or infer, between the lines, or other objects. Testing machine intelligence means verifying that the software keeps identifying the same categories and does not change them. Testing AI remains simple while no new categories are added.

If something else is being tested than categorization, interpreting test results can become quite difficult. Remember that test results should be known in advance. AI behavior is not known before executing the test.

Thus, it can happen that when evaluating test results, responsibility must turn back to humans in case the response of the test case is something else than one of AI’s established categories. Adding another category to AI is connected to relearning from scratch. You must supply all given evidence again and accept that the category borders move. In such cases, testing AI also re-starts from the beginning with establishing a new baseline. As ultimate consequence of such a worst-case scenario, the AI-driven VRS might go out of service until completely retested.

Repeat the Tests – Forever

Not only learning data changes, categories themselves are not except from change. Certain categories such as legal behavior in traffic are also subject to change and must be adapted to new environments and facts. Testing AI will detect such changes.

Therefore, for the lifetime of the AI system, testing must repeat. AI systems consist not of stable, always repeatable software but depend from their environment. If the AI system fails to reproduce correct answers, it might indicate a shift in the learning data and probably learning must restart from the beginning. Such restarts are required, for instance, if in traffic new conveyors appear, such as scooters, electro-scooters, electro-bikes, or if rules change.

An Advanced Driving Assistance System (ADAS) as an Example

The sample ADAS service we use to demonstrate the principles is a *Car Driving Function* based on a *Visual Recognition System* (VRS; Camera driven by a *Sensor Bus*) interpreting images. A *Lidar – Light Detection and Ranging,* a device that measures distances with a pulsed laser light – delivers distances and allows the *Car Driving Function* to build a 3D-model of the immediate surroundings, identifying fixed and movable objects from the image captured and analyzed. Sequences of images serve for determining the objects movements and direction.

ADAS Functionality

The *Car Driving Function* asks the *Recommender* for advice and acts in accordance with the selected route that the navigation system stored in the *Remember Routes* persistent database. This is a simplified ADAS for instructional purpose only; it possibly can power a model car. The model is equipped with camera, Lidar, and sensors for slippery roads. It uses a *Navigator* service to find a route. Both the recommender and the VRS are implemented as neural network engines. However, the system lacks the redundancy required for the real world.

The initial part of Figure 4 connects these services with sensors and car steering devices, triggered by *Look* and *Act*. The full ADAS application for our model car consists of four more parts:

- *Find Route,* e.g. by help of a navigation system, or according car user's preference.
- *Locate,* compare current location with actual route.
- *Check Route*, used to compare different possible routes in terms of traffic, weather, any other obstacles or fitting car user's preferences.
- *Amend Route*; after receiving an Alert from the *Navigator* application because conditions changed under way, changing route might help.

Finding a route is usually based on some Navigator service that can propose a route between current location and some known destinations. The complete ADAS is shown in Figure 4 as a data movement map; sizing information included in the upper left corner.

Testing the ADAS

This piece of software first prepares the setting – collecting car specifics, test cases, extending them – then executes testing first the neural network engine, then the recommender, finally the Lidar and the camera.

The testing software resides local, on the car, but the test data originate from a repository called *Testing Cloud* common to all cars undergoing the same tests. Test cases originate there, and the Testing AI engine also works on this cloud service. The ADAS of the car could upload images taken for adding those to the testing cloud; however, this is neither reflected in the part of the ADAS shown before, nor in Figure 5. Only test results are recorded in the testing cloud, upon approval by the car user, the owner of the test results. Figure 5 consists of test preparation, execution of tests for the *Neural Network,* the *Recommender,* and the *Visual Recognition Systems* including the *Lidar,* plus a test result recording and test result presentation for the tester testing the ADAS. It represents an application by itself, with user stories and the need for testing.

Figure 4. *The Complete ADAS Model*

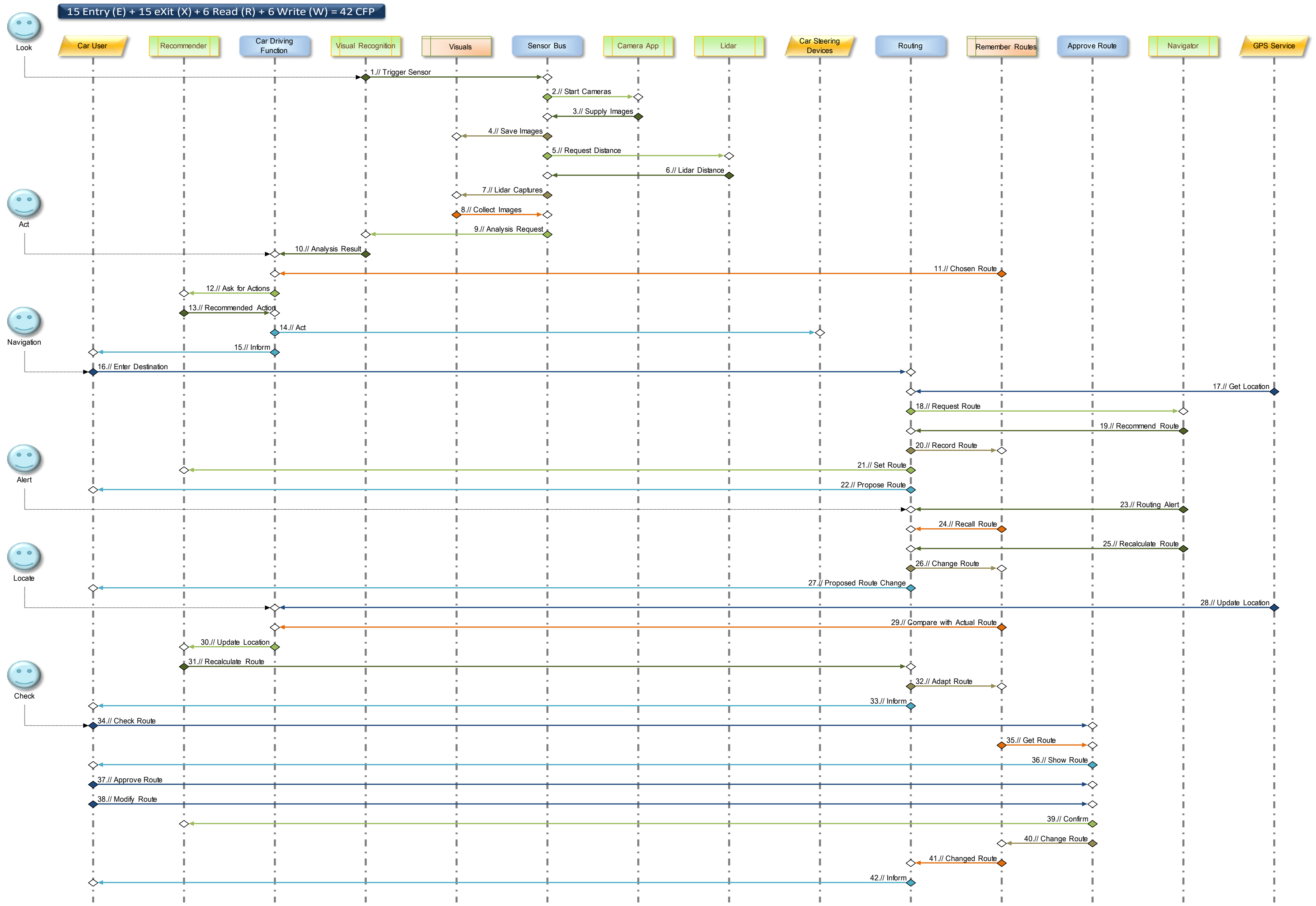
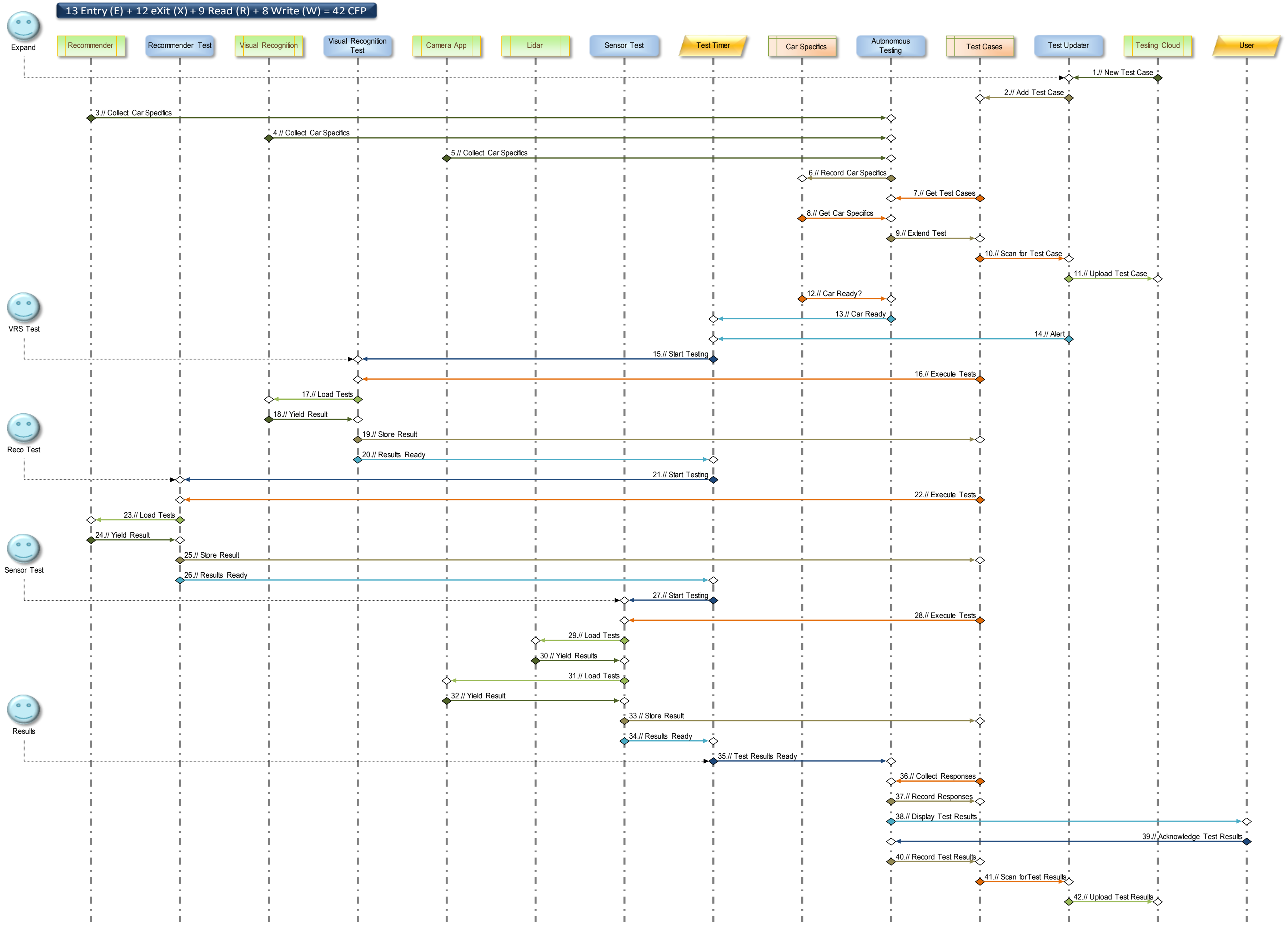


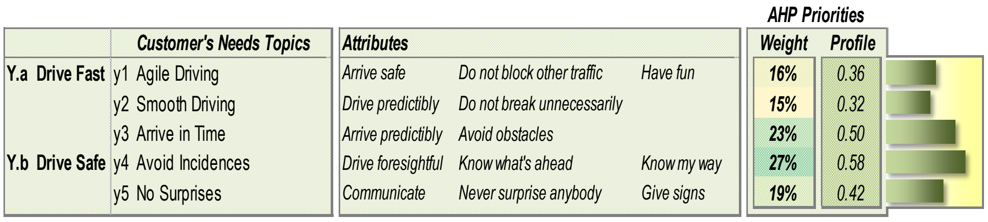
Figure 5. *Automated Real-Time Testing (ART) for Advanced Driving Assistant System (ADAS)*



The Car Users' Needs

Using the AHP, we identify the following major values for users of the ADAS as listed in Figure 6.

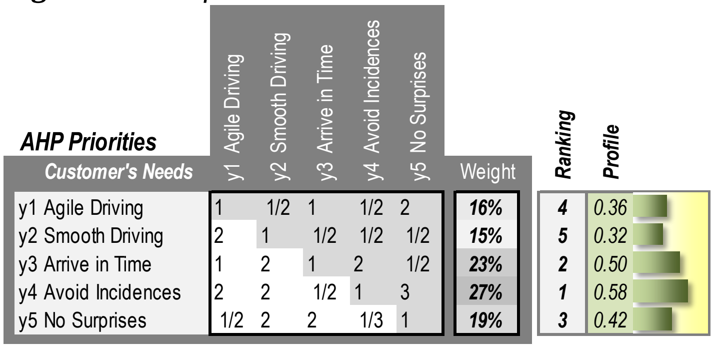
Figure 6. *Car Users' Needs*



	Customer's Needs Topics	Attributes			AHP Priorities Weight	AHP Priorities Profile
Y.a Drive Fast	y1 Agile Driving	Arrive safe	Do not block other traffic	Have fun	16%	0.36
	y2 Smooth Driving	Drive predictibly	Do not break unnecessarily		15%	0.32
	y3 Arrive in Time	Arrive predictibly	Avoid obstacles		23%	0.50
Y.b Drive Safe	y4 Avoid Incidences	Drive foresightful	Know what's ahead	Know my way	27%	0.58
	y5 No Surprises	Communicate	Never surprise anybody	Give signs	19%	0.42

The AHP process is used to analyze these needs and produce a profile for its relative importance. The profile for the car users' needs is based on the following pairwise comparison, shown in Figure 7. This is again a basic AHP.

Figure 7. *AHP for ADAS*



AHP Priorities

Customer's Needs	y1 Agile Driving	y2 Smooth Driving	y3 Arrive in Time	y4 Avoid Incidences	y5 No Surprises	Weight	Ranking	Profile
y1 Agile Driving	1	1/2	1	1/2	2	16%	4	0.36
y2 Smooth Driving	2	1	1/2	1/2	1/2	15%	5	0.32
y3 Arrive in Time	1	2	1	2	1/2	23%	2	0.50
y4 Avoid Incidences	2	2	1/2	1	3	27%	1	0.58
y5 No Surprises	1/2	2	2	1/3	1	19%	3	0.42

The needs of human drivers in today's traffic might be individually quite different; however, in view of an ADAS, characteristics linked to safety and avoidance of disturbance are dominant. You use an ADAS because you need something that helps through dense urban traffic, avoids jams and incidences, and makes driving experience smoother.

An ADAS is less suited for people who drive cars just for fun. They eventually turn it off. Their needs are not investigated by that AHP; an AHP for such people likely would produce a different car users' needs profile.

User Stories – The Functional User Requirements (FUR)

The data movements are those of the joint ADAS data movement map Figure 4. The user stories for ADAS are summarized in Table 1.

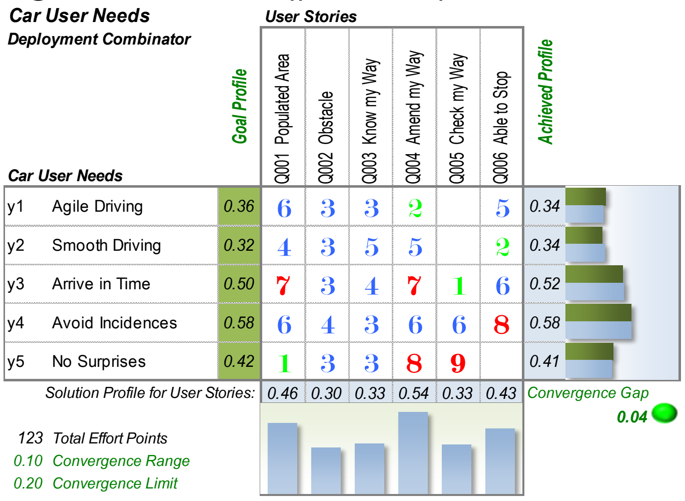
Table 1. *ADAS User Stories*

Label	**As a …**	**I want to …**	**Such that …**	**So that …**
Populated Area	Car User	let my car reduce speed	my car can safely stop	my car is not causing delays by an incidence
Obstacle	Car User	let my car avoid obstacles	my car can drive around	my car is not stopping unnecessarily
Know my Way	Car User	let my car take appropriate routes	my car avoids blocked routes and traffic jams	I know when I'll arrive
Amend my Way	Car User	optimize my route when needed	no incidence blocks my way	I still can predict when I'll arrive
Check my Way	Car User	approve or disapprove the car's choice for routing	I can take my preferred route	I feel in control
Able to Stop	Car User	have my car break soon enough	it can avoid dangerous situations	It recognizes obstacles ahead
Check my Way	Car User	approve or disapprove the car's choice for routing	I can take my preferred route	I feel in control

The user stories remain on a high epic level without specifying the details how the ADAS should behave in specific cases. With these user stories, the functional effectiveness matrix yields a satisfying rationale for the user stories (Figure 8). It means that the data movement map implements the user stories completely and without any wrong focus.

The functional effectiveness transfer function maps the user stories onto the car users' needs by counting how many data movements contribute to the user stories. This yields the cause-effect relation between functionality and requirements; also, it assigns data movements to at least one user story.

Figure 8. *Functional Effectiveness for ADAS*



Car User Needs
Deployment Combinator

User Stories

Car User Needs		Goal Profile	Q001 Populated Area	Q002 Obstacle	Q003 Know my Way	Q004 Amend my Way	Q005 Check my Way	Q006 Able to Stop	Achieved Profile
y1	Agile Driving	0.36	6	3	3	2		5	0.34
y2	Smooth Driving	0.32	4	3	5	5		2	0.34
y3	Arrive in Time	0.50	7	3	4	7	1	6	0.52
y4	Avoid Incidences	0.58	6	4	3	6	6	8	0.58
y5	No Surprises	0.42	1	3	3	8	9		0.41
Solution Profile for User Stories:			0.46	0.30	0.33	0.54	0.33	0.43	

Convergence Gap 0.04 ●

123 Total Effort Points
0.10 Convergence Range
0.20 Convergence Limit

The Test Stories

The test stories in Table 2 specify more details how to implement the ADAS.

Table 2. *Test Stories for ADAS for User Stories Shown in Table 1*

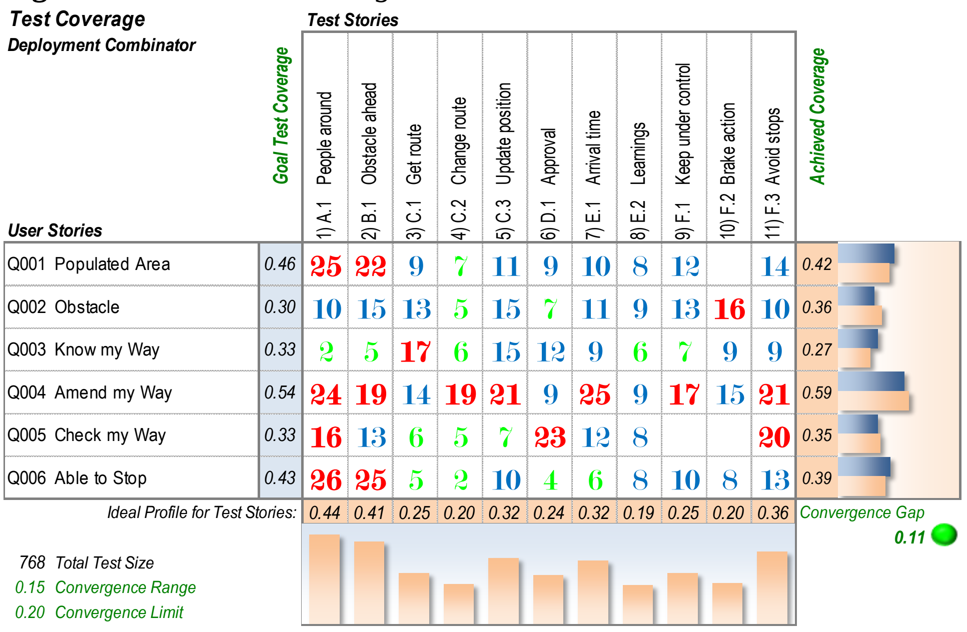
			Test Story	**Informal Description**
A	People Around	A.1	People around	The ADAS identifies people staying near the car, or sit on a specific vehicle
B	Obstacle	B.1	Obstacle ahead	Static or moving obstacles I the way are identified and correctly classified
C	Know my way	C.1	Get route	The ADAS always knows where to go next, be it at a crossing or at the end of the route
		C.2	Change route	The route to take is periodically changed, based on alerts received from *Navigator*
		C.3	Update position	The correct position is always known to the ADAS
D	Choose way	D.1	Approval	When choosing the route, the car user must approve the decision
		E.1	Arrival time	The expected arrival time is shown to the car user
		E.2	Learnings	The ADAS has a repository of routes taken and can rely on past experiences, e.g. jams
		F.1	Keep under control	The ADAS is always in control of the car, even if he car user intervenes
		F.2	Brake action	Brake action is known to ADAS, depending on weather and road condition
		F.3	Avoid stops	The ADAS tries to drive smoothly, adapting speed, avoiding unnecessary stops

Remember that we had no clue how our visual recognition system determines the list of valid objects that it recognizes. Possibly it is implemented as a neural network, or a *Support Vector Machine* (SVM) is used (Pupale 2018). Nevertheless, we use our data movement map model to assess functional effectiveness with the later goal of testing it. In other words: we test what we think how the VRS works. We test our model.

For this paper, we give only informal descriptions, leaving it to the reader to invent suitable test cases. For the calculation below, we used around five test cases per test story. This yields the following test coverage (Figure 9).

The numbers in the cell represent the number of data movements that support the respective user story. With a convergence gap of 0.11 we are within convergence range, set a bit wider than in usual transfer functions.

Figure 9. *Initial Test Coverage*



Test Coverage
Deployment Combinator

User Stories	Goal Test Coverage	1) A.1 People around	2) B.1 Obstacle ahead	3) C.1 Get route	4) C.2 Change route	5) C.3 Update position	6) D.1 Approval	7) E.1 Arrival time	8) E.2 Learnings	9) F.1 Keep under control	10) F.2 Brake action	11) F.3 Avoid stops	Achieved Coverage
Q001 Populated Area	0.46	25	22	9	7	11	9	10	8	12		14	0.42
Q002 Obstacle	0.30	10	15	13	5	15	7	11	9	13	16	10	0.36
Q003 Know my Way	0.33	2	5	17	6	15	12	9	6	7	9	9	0.27
Q004 Amend my Way	0.54	24	19	14	19	21	9	25	9	17	15	21	0.59
Q005 Check my Way	0.33	16	13	6	5	7	23	12	8			20	0.35
Q006 Able to Stop	0.43	26	25	5	2	10	4	6	8	10	8	13	0.39
Ideal Profile for Test Stories:		0.44	0.41	0.25	0.20	0.32	0.24	0.32	0.19	0.25	0.20	0.36	Convergence Gap 0.11

768 Total Test Size
0.15 Convergence Range
0.20 Convergence Limit

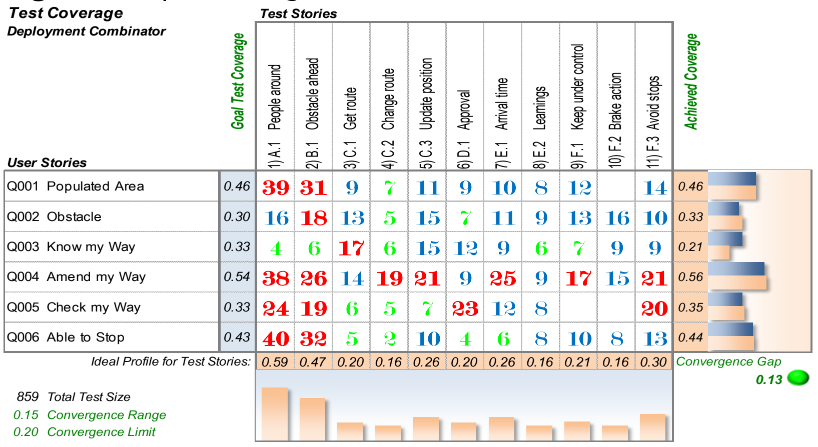
Extending Test Cases

Extending test cases within the same test stories yields more reliable results, and a higher test intensity (see Figure 13). In this example, extension works in two stages:

- Adding test cases that refer to bad weather forecast. If the *Navigator* reports rain on the route, driving speed and arrival forecast must be adapted.
- More test cases are added after the *Navigator* reports stormy weather causing eventually a change to the chosen route.

The following matrices (Figures 10 and 11) show the results after each of the two steps outlined above.

Figure 10. *After Adding Bad Weather Cases*

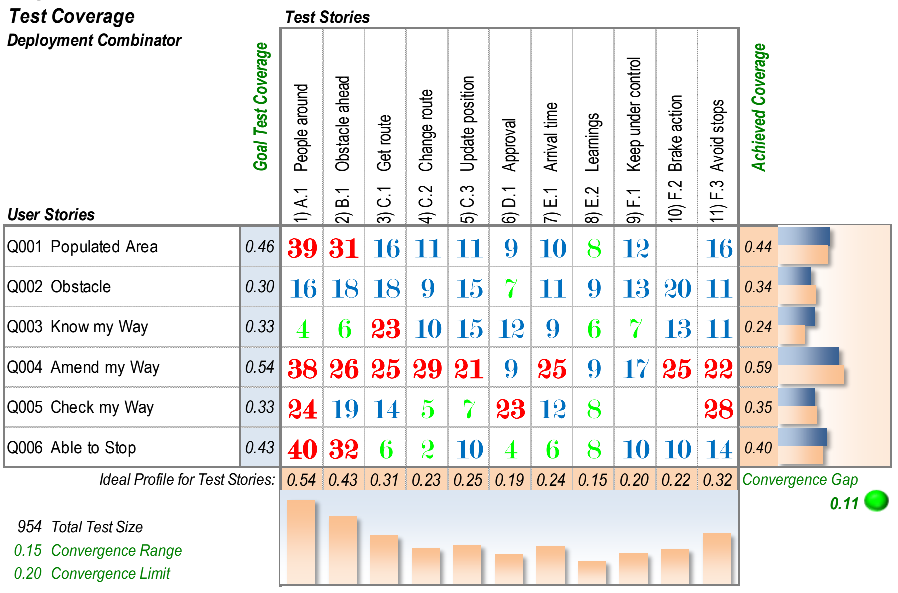


Test Coverage
Deployment Combinator

User Stories	Goal Test Coverage	1) A.1 People around	2) B.1 Obstacle ahead	3) C.1 Get route	4) C.2 Change route	5) C.3 Update position	6) D.1 Approval	7) E.1 Arrival time	8) E.2 Learnings	9) F.1 Keep under control	10) F.2 Brake action	11) F.3 Avoid stops	Achieved Coverage
Q001 Populated Area	0.46	39	31	9	7	11	9	10	8	12		14	0.46
Q002 Obstacle	0.30	16	18	13	5	15	7	11	9	13	16	10	0.33
Q003 Know my Way	0.33	4	6	17	6	15	12	9	6	7	9	9	0.21
Q004 Amend my Way	0.54	38	26	14	19	21	9	25	9	17	15	21	0.56
Q005 Check my Way	0.33	24	19	6	5	7	23	12	8			20	0.35
Q006 Able to Stop	0.43	40	32	5	2	10	4	6	8	10	8	13	0.44
Ideal Profile for Test Stories:		0.59	0.47	0.20	0.16	0.26	0.20	0.26	0.16	0.21	0.16	0.30	Convergence Gap 0.13

859 Total Test Size
0.15 Convergence Range
0.20 Convergence Limit

Figure 11. *After Adding Adaption to Changed Route*



Test Coverage
Deployment Combinator

User Stories	Goal Test Coverage	1) A.1 People around	2) B.1 Obstacle ahead	3) C.1 Get route	4) C.2 Change route	5) C.3 Update position	6) D.1 Approval	7) E.1 Arrival time	8) E.2 Learnings	9) F.1 Keep under control	10) F.2 Brake action	11) F.3 Avoid stops	Achieved Coverage
Q001 Populated Area	0.46	39	31	16	11	11	9	10	8	12		16	0.44
Q002 Obstacle	0.30	16	18	18	9	15	7	11	9	13	20	11	0.34
Q003 Know my Way	0.33	4	6	23	10	15	12	9	6	7	13	11	0.24
Q004 Amend my Way	0.54	38	26	25	29	21	9	25	9	17	25	22	0.59
Q005 Check my Way	0.33	24	19	14	5	7	23	12	8			28	0.35
Q006 Able to Stop	0.43	40	32	6	2	10	4	6	8	10	10	14	0.40
Ideal Profile for Test Stories:		0.54	0.43	0.31	0.23	0.25	0.19	0.24	0.15	0.20	0.22	0.32	Convergence Gap 0.11 ●

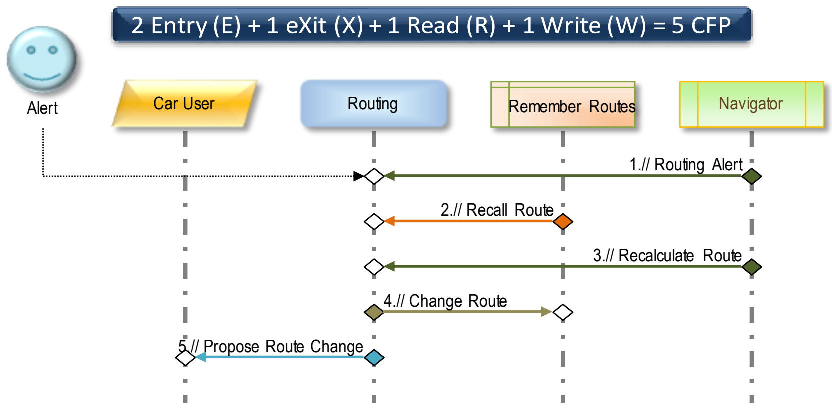
954 Total Test Size
0.15 Convergence Range
0.20 Convergence Limit

ART detects these new test cases because the data group received from the *Navigator* contains a weather forecast, as part of the route description. New test cases are created starting from the existing ones, by variation of test data, considering other all data received from data movements. Obviously, weather forecast changes the driving time prediction. Among the many test cases that can be created, ART keeps the convergence gap within limits, using this as selection process. Total test size is growing, and convergence gap is stable, or shrinking. This is the benefit of using customers' needs as goal for testing.

How Came the Weather Forecast into ART?

The additional test cases improve reliability and accuracy. ART finds such extensions by scanning data groups of the data movements involved. Since the chosen route is not fix but changes on receiving an *Alert* from the *Navigator*, ART learns that conditions such as rainy and stormy weather can exist and generates suitable test cases; shown in data movement map Figure 12.

Figure 12. *Amend Route upon Navigator's Alert*



The data group moved by the data movement *Routing Alert* from the *Navigator* application to the *Routing* functional process contains all sort of alerts, including traffic jams and bad weather conditions. The ART mechanism extending test cases considers weather as a reason to change driving. Thus, when replacing other reasons for choosing a route, the *Chosen Route* data movement in *Look & Act* (initial two triggers in Figure 4) tells the *Car Driving Function* about the changed weather conditions. This attribute is now selectable by ART for generating new test cases, also for the *Visual Recognition* system (VRS). Thus, it will be added as another test case for VRS, sooner or later. And because the new test case fits well with the car users' needs, rather sooner than later.

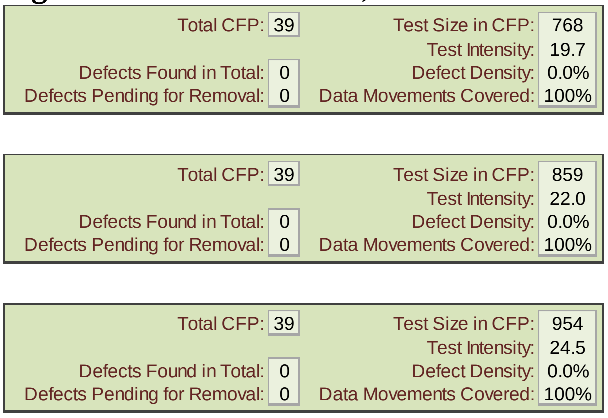
ART thus must find images showing people, or other vehicles, in the rain, or in a storm, to produce the same results in the test stories *A.1: People around*; *B.1: Obstacle ahead*; *C.1: Get route*; and *C.2: Change route*.

Weather is one thing that can be considered. But there is much more before autonomous cars can hit the road, for instance a tendency, or the need, to use bikes for transporting bags in certain social environments. Moreover, where Ms. Herzberg crossed, there is a functional road strip across the median of the four-lane road with the potential of being abused by pedestrians. If the *Navigator* could also consider additional information about the neighborhoods traversed, Elaine Herzberg possibly would still be alive, and trust in autonomous cars unhampered.

Summary View

The summary view on the original and the two extended test suites reveals, as expected, that test size and intensity increased, while we might expect more defects detected after the tests were executed.

Figure 13. *Initial Test Suite, and Two Extensions*



Total CFP:	39	Test Size in CFP:	768
		Test Intensity:	19.7
Defects Found in Total:	0	Defect Density:	0.0%
Defects Pending for Removal:	0	Data Movements Covered:	100%

Total CFP:	39	Test Size in CFP:	859
		Test Intensity:	22.0
Defects Found in Total:	0	Defect Density:	0.0%
Defects Pending for Removal:	0	Data Movements Covered:	100%

Total CFP:	39	Test Size in CFP:	954
		Test Intensity:	24.5
Defects Found in Total:	0	Defect Density:	0.0%
Defects Pending for Removal:	0	Data Movements Covered:	100%

Functional size remained stable: CFP 39, while increasing test size also increased test intensity.

Thus, improving testing is always possible by simply extending the test cases by similar ones, provided test coverage keeps the convergence gap narrow enough. ART provides value without increasing functional size. In this example, it was enough to trace back data movements that could contribute extra data to tests. Thus, the data movement map is paramount for automatic test case generation.

For testers, it is enough to provide an initial test suite (Table 2). The rest is left to AI. So, you test AI with AI. You can increase test intensity as much as you like, and our budget allows More tests certainly increase opportunities for detecting defects that can be removed and add marketable value for the customer. Thanks to the test coverage transfer function and its convergence gap, those additional tests remain relevant. Moreover, since tests are generated randomly, there is no bias blocking certain test cases, although extending test cases along some application cases such as weather or route change might allow for targeted test extensions.

The Next Steps, and a Preliminary Conclusion

The basic idea how to deal with "untestable" neuronal networks and deep learning is to create a data movement map as a model, specifying what users expect the AI part to do.

Clearly, a VRS needs more tests than fitting in this paper. ART generates more test cases out of the fourteen test stories to increate test intensity. However, at the current stage of research, we have no clue what test intensity is enough for a VRS in an autonomous car.

The proposed method can be upscaled to larger test coverage transfer functions. Real software systems have a few hundred user stories, and even more test stories. Solving such test coverage matrices requires big data algorithms but these tools are readily available nowadays.

A promising approach is to use the AHP for initial test coverage. With many user stories and test stories, a complex system often splits into smaller parts than can be tested separately, at least initially, and the full test coverage matrix fills in automatically by ART (Fehlmann 2019).

Applying ART means adding more test cases, more image sequences, always with respect to the convergence gap, aiming at improving the convergence gap. This limits combinatorial explosion, as it allows selecting relevant test cases only.

References

Bell D (2004) *UML basics: the sequence diagram – introductory level.* Armonk, NY: IBM DeveloperWorks.

Fehlmann TM (2016) *Managing complexity - uncover the mysteries with six sigma transfer functions.* Berlin, Germany: Logos Press.

Fehlmann TM (2019) *Autonomous real-time testing - testing artificial intelligence and other complex systems.* Berlin, Germany: Logos Press.

Fehlmann TM, Kranich E (2014) *Uncovering customer needs from net promoter scores.* Istanbul, Turkey: 20th International Symposium on Quality Function Deployment.

Fehlmann TM, Kranich E (2017) *Autonomous real-time software & systems testing.* Göteborg: s.n.

Fehlmann TM, Kranich E (2018) Theoretical aspects of consumer metrics for safety & privacy. In X Larrucea, I Santamaria, R O'Connor, R Messnarz (eds), *Systems, Software and Services Process Improvement. EuroSPI 2018. Communications in Computer and Information Science*, 649-653. Springer, Cham.

Gerven Mv, Bohte S (2017) *Artificial neural networks as models of neural information processing.* Lausanne: Frontiers Media.

ISO 16355-1:2015 (2015) *Applications of statistical and related methods to new technology and product development process - part 1: general principles and perspectives of quality function deployment (QFD).* Geneva, Switzerland: ISO TC 69/SC 8/WG 2 N 14.

ISO/IEC 19761 (2011) *Software engineering - COSMIC: a functional size measurement method.* Geneva, Switzerland: ISO/IEC JTC 1/SC 7.

ISO/IEC/IEEE 29119-4 (2015) *Software and systems engineering — software testing — part 4: test techniques.* Geneva, Switzerland: ISO/IEC JTC 1.

Mazur G (2014) *QFD and the new voice of customer (VOC).* Istanbul, Turkey: International Council for QFD (ICQFD), 13-26.

Mazur G, Bylund N (2009) *Globalizing gemba visits for multinationals.* Savannah, GA, USA: Transactions from the 21st Symposium on Quality Function Deployment.

Pupale R (2018) *Support vector machines (svm) - an overview.* Retrieved from: https://bit.ly/36KQkRD. [Accessed 28 March 2019].

Saaty TL (2003) Decision-making with the AHP: why is the principal eigenvector necessary? *European Journal of Operational Research* 145(1): 85-91.

Schwaber K, Beedle M (2002) *Agile software development with scrum.* Upper Saddle River, NJ: Prentice Hall PTR.

The National Transportation Safety Board (2018) *Preliminary Report Highway Hwy18mh010.* Retrieved from: https://bit.ly/2NtKlZO. [Accessed 13 March 2019].

van Gerven M, Bothe S (2018) *Artificial neural networks as models of neural information processing.* Lausanne: s.n.