Intuitionism and Computer Science – Why Computer Scientists do not Like the Axiom of Choice

Dr. Thomas Fehlmann
Euro Project Office AG
8049 Zürich, Switzerland
+41 44 253 1306
thomas.fehlmann@e-p-o.com

Eberhard Kranich
Euro Project Office
47051 Duisburg, Germany
+41 44 253 1306
eberhard.kranich@e-p-o.com

Abstract

The Axiom of Choice (AC) says that every set has a representative element. However, deterministic computers cannot produce arbitrary elements. They need some algorithm that tells them, which one to choose. But then, the element is no longer arbitrary. Even for a true random generator, you will need Entropy. This is data gathered from outside the system, and we as Theoretical Computer Scientists do not like that. Thus, we need to understand the Axiom of Choice better. For this, we use a model of Combinatory Logic.

Keywords: Combinatory Logic, Combinatory Algebra, Intuitionism, Axiom of Choice, Computability, Software Testing, Artificial Intelligence.

Introduction

Zermelo–Fraenkel (ZF) set theory, named after mathematicians Ernst Zermelo and Abraham Fraenkel, is an axiomatic system that was proposed in the early twentieth century to formulate a theory of sets free of paradoxes such as Russell's paradox. For an introduction, see e.g., Potter (Potter, 2004).

The famous *Banach–Tarski paradox* is a theorem in set-theoretic geometry, which states the following: Given a solid sphere in 3-dimensional space, there exists a decomposition of the sphere into a finite number of disjoint subsets, which can then be put back together in a different way to yield two identical copies of the original sphere. Indeed, the reassembly process involves only moving the pieces around and rotating them without changing their shape. (Banach & Tarski, 1924).

The number of pieces was subsequently reduced to five by Robinson (Robinson, 1947), although the pieces are extremely complicated. Five pieces are minimal, although four pieces are enough if the single point at the center is neglected.

This sounds strange, counter-intuitive, and impracticable. Nevertheless, it relies on the following reasoning:

- If the Axiom of Choice (AC) holds, then non-measurable sets exist (Tao, 2011);
- If non-measurable sets exist, the Banach-Tarski paradox holds (Pawlikowski, 1991).

A set is called *measurable*, if there is a systematic way to assign a number to each suitable subset, a *Size*, such that sizes of subsets can be added to get a measure for the size of the original set. For details, see Potter (Potter, 2004).

So, why can the Banach-Tarski paradox be proven to be true, logically? Is mathematical logic flawed? No: A formal proof of the paradox uses an infinity of spheres (set of all points whose distance to origin is constant), excluding the spheres of radius 0. For an elegant proof, see Rauglaudre (Rauglaudre, 2017). For a quick idea instead of a full proof, just imagine that you always can fill your sphere with infinitely many copies of that sphere, pick a point in each sphere, rotate them by an angle to get yet another new sphere but different from all previously picked spheres – this arguments uses the AC! – and still get infinitely many copies. Now separate the original set of spheres from the copies and you have two different, identical spheres.

The Axiom of Choice and its Variants

The Axiom of Choice (AC) says that given any family of non-empty sets S_i for $i \in I$, there exists a function such that $f(i) \in S_i$ for all $i \in I$. f is called a *Choice Function*.

Obviously, the Banach-Tarski paradox makes it difficult to believe that the AC is indispensable in mathematics. Equivalent to the AC is the *Well-ordering Theorem*. It states that every set can be well-ordered. A set X is well-ordered by a strict total order if every non-empty subset of X has a least element under the ordering. This is intuitively not compelling, too.

Real, Irrational Numbers Require the AC

However, the AC is indispensable for many important – and intuitive – mathematical results; among them

- Let $\mathbb{R}$ be the closure of $\mathbb{Q}$, the set of all rational numbers under convergent sequences. Then, the convergence point is also in $\mathbb{R}$.
- Many square numbers, such as $\sqrt{2}$, are not rational numbers, since assuming there are natural numbers $p, q \in \mathbb{N}$ with $\sqrt{2} = p/q$ leads to the conclusion that both p, q must be dividable by 2. This contradicts the possibility of representing rational numbers by co-primes[1].

Nevertheless, irrational numbers are unhandy for a digital device. You only can represent them by their properties, i.e., as symbols.

Combinatory Logic is a notation to eliminate the need for quantified variables in mathematical logic. The issue addressed with combinatory logic is the AC. What means "there exists something", $\exists x \in M$, in some set M? Informally, the AC says that given any collection of non-empty sets, it is possible to select exactly one object from each set, without requiring an algorithm saying how the selection is done. In the theory of *Complex Analysis*, such an algorithm

[1] Co-primes are numbers which have 1 as the greatest common denominator.

seems an unnecessary condition; in fact, complex analysis proved to be very successful without requiring constructive selection algorithms.

The Intuitionistic Variant of the Axiom of Choice

In Computer Science, the existence of selection algorithms seems a natural condition for the applicability of the AC. On a computer, nothing exists resembling a program or process without an algorithm that effectively constructs it.

Thus, as computer scientists we always presume a stronger version of the AC: there exists means, there exists an algorithm that allows to select exactly one representative from each collection of sets.

Interestingly, this conditioning of the AC to Mathematical Logic has wide consequences. For instance, there exists a countable model of the real numbers, meeting all the axioms for real numbers. We can assure that the limit of any convergent sequence of real numbers exists and that it is itself a real number by selecting the convergence sequences themselves as a model. This is a measurable and enumerable set. In fact, there is no other way on a computer to implement real numbers than by such sequences.

The famous digital representation for the relation between diameter and circumference of a circle, π, is an infinite sequence of digits that never repeat themselves; thus, not anything that exists within a digital device, not even within the universe. Only sequences that converge towards π do exist.

However, the Banach–Tarski paradox does not hold with the intuitionistic version of the AC, since there is no way selecting the right rotated spheres that allow to split the original ball into two identical spheres.

Combinatory Logic

There is a mathematical theory about *Combinatory Algebras* (Engeler, 1995) that explains quite generally how to combine topics in areas of knowledge. Combination is not only on the basic level possible; you can also explain how to combine topics on the second level; sometimes called meta-level. Intuitively, we would expect such a meta-level describing knowledge about how to deal with different knowledge areas.

Combinatory algebras are models of *Combinatory Logic* (Curry & Feys, 1958) and (Curry, et al., 1972). These are algebras that are combinatory complete; i.e., there is a combination operation $M \bullet N$ for all elements M, N in the combinatory algebra and the following two *Combinators* **S** and **K** can be defined with the following properties

$$\mathbf{K} \bullet P \bullet Q = P \tag{1}$$

and

$$\mathbf{S} \bullet P \bullet Q \bullet R = P \bullet Q \bullet (P \bullet R) \tag{2}$$

where P, Q, R are elements in the combinatory algebra.

Thus, the combinator **K** acts as projection, and **S** is a substitution operator for terms in the combinatory algebra. Like an assembly language, the **S**-**K** terms become quite lengthy and are barely readable by humans, but they work fine as a foundation for computer science.

The power of these two operators is best understood when we use them to define other, handier, and more understandable combinators:

Identity
The identity combinator is defined as

$$\mathbf{I} \coloneqq \mathbf{S} \bullet \mathbf{K} \bullet \mathbf{K} \tag{3}$$

Indeed, $\mathbf{I} \bullet M = \mathbf{S} \bullet \mathbf{K} \bullet \mathbf{K} \bullet M = \mathbf{K} \bullet M \bullet (\mathbf{K} \bullet M) = M$. Association is to the left.

Functionality by the Lambda Combinator
Curry's *Lambda Calculus* (Barendregt, 1977) is a formal language that can be understood as a prototype programming language.

The algebra of **S**-**K** terms models the lambda calculus by recursively defining the *Lambda Combinator* for a variable **x** as follows:

$$\begin{gathered} \mathbf{L_x} \bullet x = \mathbf{I} \\ \mathbf{L_x} \bullet Y = \mathbf{K} \bullet Y \text{ if } Y \text{ different from } \mathbf{x} \\ \mathbf{L_x} \bullet M \bullet N = \mathbf{S} \bullet \mathbf{L_x} \bullet M \bullet \mathbf{L_x} \bullet N \end{gathered} \tag{4}$$

The definition holds for any variable **x** in the combinatory algebra.

For more details about the foundations of Mathematical Logic, see for instance Barwise (Barwise, et al., 1977) or Potter (Potter, 2004). For more combinators in combinatory logic, see e.g., Zachos (Zachos, 1978).

Arrow Terms

Let $\mathcal{L}$ be the set of all assertions over a given domain. Examples include statements about customer's needs, solution characteristics, methods used, program states, test conditions, etc. These statements are assertions about the business domain we are dealing with.

An *Arrow Term* is recursively defined as follows:

- Every element of $\mathcal{L}$ is an arrow term
- Let $a_1, \dots, a_m, b$ be arrow terms. Then

$$\{a_1, \dots, a_m\} \rightarrow b \tag{5}$$

is also an arrow term.

The left-hand side of an arrow term is a finite set of arrow terms and the right-hand side is a single arrow term. This definition is recursive. The arrows might suggest cause-effect, not logical imply.

The Algebra of Arrow Terms

Denote by $\mathcal{G}(\mathcal{L})$ the power set containing all *Arrow Terms* of the form (5). The formal, recursive, definition, in set-theoretical language, is given in equation (6):

$$\begin{gathered}\mathcal{G}_0(\mathcal{L}) = \mathcal{L} \\ \mathcal{G}_{n+1}(\mathcal{L}) = \\ \mathcal{G}_n(\mathcal{L}) \cup \left\{\{a_1, \dots, a_m\} \to b \middle| a_1, \dots, a_m, b \in G_n(L), m \in \mathbb{N}\right\}\end{gathered} \tag{6}$$

for $n = 0, 1, 2, \dots$ $\mathcal{G}(\mathcal{L})$ is the set of all (finite and infinite) subsets of the union of all $\mathcal{G}_n(\mathcal{L})$:

$$\mathcal{G}(\mathcal{L}) = \bigcup_{n \in \mathbb{N}} \mathcal{G}_n(\mathcal{L}) \tag{7}$$

The elements of $\mathcal{G}_n(\mathcal{L})$ are arrow terms of level n. Terms of level 0 are *Assertions*, terms of level 1 *Rules*. A set of rules is called *Rule Set* (Fehlmann, 2016). In general, a rule set is a finite set of arrow terms. We call infinite rule sets a *Knowledge Base*. Hence, knowledge is a potentially unlimited set of rules sets containing rules about assertions regarding our domain.

Combining Rule Sets

We can combine two rule sets as follows:

$$M \bullet N = \{c | \exists \{b_1, b_2, \dots, b_m\} \to c \in M;\ b_i \in N\} \tag{8}$$

Arrow Term Notation

To avoid the many set-theoretical parenthesis, the following notations, that we call *Arrow Schemes*, are applied:

- a_i for a finite set of arrow terms, i denoting some finite indexing function for arrow terms.
- a_1 for a singleton set of arrow terms; i.e. $a_1 = \{a\}$ where a is an arrow term.
- $\emptyset$ for the empty set, such as in the arrow term $\emptyset \to a$.
- $a_i + b_j$ for the union of two sets a_i and b_j of arrow terms.

The indexing function cascades, thus $a_{i,j}$ denotes the union of a finite number of m arrow term sets

$$a_{i,j} = a_{i,1} \cup a_{i,2} \cup \dots \cup a_{i,j} \cup \dots \cup a_{i,m} = \bigcup_{k=1}^{m} a_{i,k} \tag{9}$$

An arrow scheme always represents a finite or infinite set of arrow terms.

With these conventions, $(x_i \to y)_j$ denotes a rule set, i.e., a non-empty finite set of arrow terms, together with two indexing functions i, j. Each set has at least one arrow. Thus, such set is of level 1 or higher.

With this notation, the application rule for M and N now reads

$$M \bullet N = (b_i \to a) \bullet b_i = \{a | \exists b_i \to a \in M;\ b_i \subset N\} \tag{10}$$

Arrow Terms – A Model of Combinatory Logic

The algebra of arrow terms is a combinatory algebra and thus a model of combinatory logic.

The following definitions demonstrate how arrow terms implement the combinators **S** and **K** fulfilling equations (1) and (2).

- $\mathbf{I} = a_1 \to a$ is the *Identification*; i.e. $(a_1 \to a) \bullet b = b$
- $\mathbf{K} = a_1 \to \emptyset \to a$ selects the 1st argument:
  $\mathbf{K} \bullet\ b \bullet \mathrm{c} = (b_1 \to \emptyset \to b) \bullet b \bullet \mathrm{c} = (\emptyset \to b) \bullet \mathrm{c} = \mathrm{b}$
- $\mathbf{KI} = \emptyset \to a_1 \to a$ selects the 2nd argument:
  $\mathbf{KI}\ \bullet b \bullet \mathrm{c} = (\emptyset \to c_1 \to c) \bullet b \bullet \mathrm{c} = (c_1 \to c) \bullet c = \mathrm{c}$
- $\mathbf{S} = \left(a_i \to \left(b_j \to c\right)\right)_1 \to (d_k \to b)_i \to \left(a_i + b_{j,i} \to c\right)$

Therefore, the algebra of arrow terms is a model of combinatory logic.

The proof that the arrow terms' definition of **S** fulfils equation (2) is somewhat more complex. The interested reader can find it in Engeler (Engeler, 1981, p. 389). With **S** and **K,** an abstraction operator can be constructed that builds new knowledge bases. This is the *Lambda Theorem*; it is proved along the same way as Barendregt's Lambda combinator (Barendregt, 1977) and (Fehlmann, 1981, p. 37).

Neural Algebra

Engeler uses the arrow terms for a brain model (Engeler, 2019). A directed graph, together with a firing law at all its nodes, constitute the connective basis of the brain model. The model itself is built on this basis by identifying brain functions with parts of the firing history. Its elements may be visualized as a directed graph, whose nodes indicate the firing of a neuron. *Cascades* describe firing between nodes (neurons) and is represented by arrow terms $\{a_1, \dots, a_m\} \to b$ where $a_1, \dots, a_m$ are sub-cascades, while the right sub-cascade *b* describes the characteristic leave of its firing history graph. The Neural Algebra is defined as a collection of subsets of the set of cascades. With the application rule (10), we have an algebraic structure.

Within this setting, it is possible to define models for reasoning and problem solving. However, not only flat reasoning, also the control operations. This is represented as a solution X for the control problem $\boldsymbol{C} \bullet X = X$, where $\boldsymbol{C}$ is the *Controlling Operator*. Engeler presents in an elegant way combinators that represent basic operations of the brain, e.g., problem solving, or balancing on a bike.

Discrimination between choices, and self-reflection about how to take decisions, how to address problems, as well as learning and comprehending can also be modeled with that approach.

Since the number of cascades that a brain can produce is finite and limited – by the lifespan of the brain – solution to the fixpoint control problems turn out to be finite cascades. It is tempting to identify cascades with though processes.

The Algebra of Test Cases

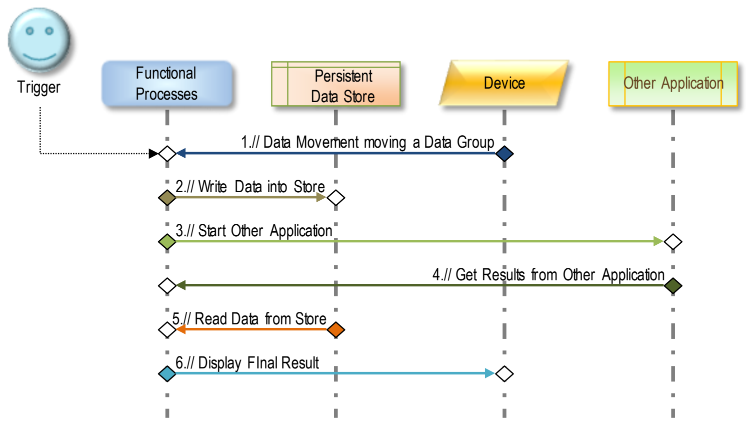
Test cases are a mapping of arrow terms onto data movement maps, see below. The data movements induce a sizing valuation on this algebra by counting the number of data movements executed once per test case. We rely on the ISO standard 19761 COSMIC (ISO/IEC 19761, 2011).

Data Movement Maps

Data Movement Maps are a way to model a piece of software by connecting objects of interest, representing functionality, persistent stores, devices, and other applications, based on the COSMIC standard. The connectors represent *Data Movements*. They have some resemblance to *UML Sequence Diagrams* (Bell, D., 2004) but with less detail; thus, without guards, loops, and alternative fragments. Also, sequencing is not prescribed.

Every data movements moves a *Data Group*, which can be thought as a data record moving information from one object of interest to another. Usually, its uniqueness is indicated by color-filled trapezes (Figure 1). Another move of same data group between the same objects within a functional process lets the trapeze blank. The number of unique movements is called *Functional Size* according COSMIC and denoted by $\|\mathfrak{s}\|$, for any data movement map $\mathfrak{s} \in \mathfrak{S}$ where $\mathfrak{S}$ is a set of data movement maps (COSMIC Measurement Practices Committee, 2017).

Figure 1: Sample Data Movement Map


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

Data movement maps are explained in (Fehlmann, 2020, p. 27) and in more detail in (Fehlmann, 2016, p. 155).

The Combinatory Algebra of Test Cases

Arrow terms over the language of test assertions, or program states, represent test cases in a straightforward way. In formula (5), the left-hand side of the arrow term $\{a_1, \dots, a_m\} \rightarrow b$ represents test data $a_1, \dots, a_m$ as a sequence of program states, while the right-hand side b is the expected resulting program state after executing the test case. Let $\mathfrak{S}$ be a finite set of data movement maps. A test case $\{a_1, \dots, a_m\} \rightarrow b$ can be *executed* in $\mathfrak{S}$ if a data movement map in $\mathfrak{S}$ exists that transforms the program states $a_1, \dots, a_m$ into b.

Denote by $\bigcup \mathfrak{S}$ the union of all data movement maps in $\mathfrak{S}$. The *union* is defined in the straightforward manner by identifying all identical objects of interest within all data movement maps in $\mathfrak{S}$. Obviously, $\bigcup \mathfrak{S}$ is itself a data movement map. It represents the program under test, or more exactly, the part of the program that is covered by test cases, executable in $\mathfrak{S}$. Note that when combining executable test cases from program $\bigcup \mathfrak{S}$ using equation (10), the result is also executable in $\bigcup \mathfrak{S}$.

Test Automation

The arrow terms serve primarily as a grammar for test cases, but the properties of a combinatory algebra allow for much more. Test can be combined, using equation (8) or any other combinator. This allows to generate as many test cases as we want and need for achieving full test coverage.

Therefore, it is no longer an excuse for not testing large and complex systems that the scarcity of resources, especially proficient software testers, do not allow for a full test, testing all of the software even for large systems such as today's trainsets, or *Advanced Driving Assistance Systems* (ADAS), or autonomous vehicles, in case they ever will hit our roads.

It is noteworthy that programmers who want to set up test concatenation $M \bullet N$ for automatic testing, need access to the test cases in N that provide the responses needed for M. Otherwise, combining tests is unsafe or cannot be automated. The equation (8) does exactly this, providing the existence of an arrow term means that a rule is available that tells the programmer, which test case to take. In other words, combination of tests also traces the history how these tests have emerged. This allows to validate combined tests.

This interpretation of the logical existence means that we apply here the intuitionistic variant of the axiom of choice. Programmers, and even more: testers, would reject combining tests with equation (8) unless we silently apply the intuitionistic, stronger form of the AC.

Keeping the Number of Generated Test Cases Low

However, a testing environment that produces test cases without end is not very practical either. It is therefore necessary to have a selection algorithm that allows to direct the test case generator towards the relevant tests.

This can be achieved by means of *Transfer Functions*, which are in detail explained in Fehlmann (Fehlmann, 2016) – that map the selection of test cases back onto customer values.

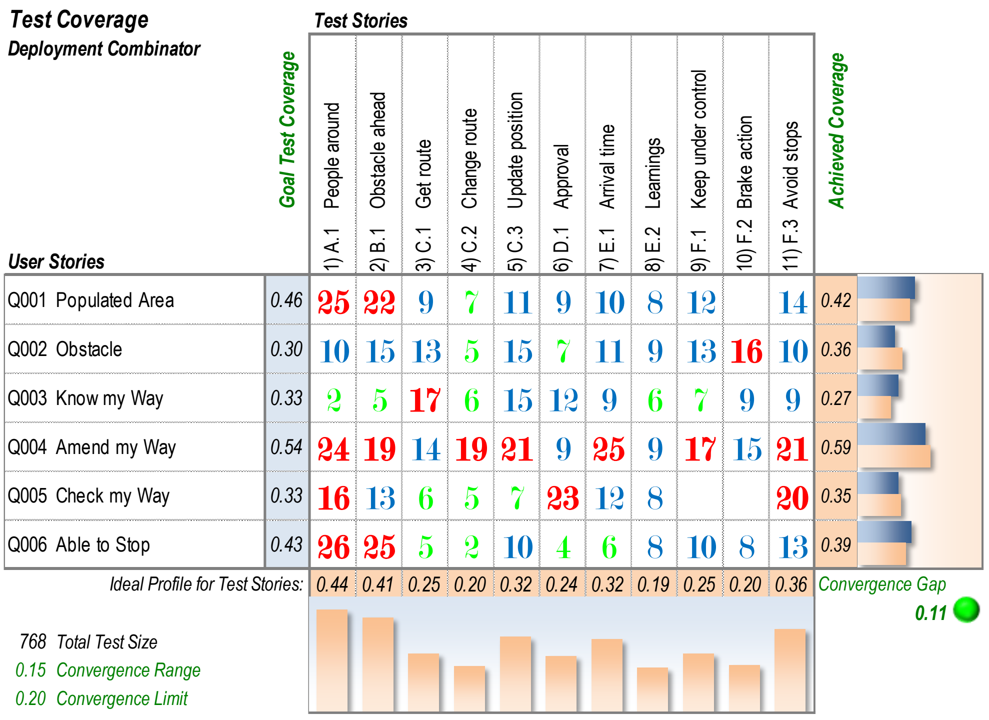
In this paper the so-called linear multiple-response transfer functions are of special interest. Such transfer functions are defined by equations of the form

$$\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x} \tag{11}$$

where $\boldsymbol{y}$ is the predefined goal profile, $\boldsymbol{A}$ as transfer function is a matrix and hence a linear transfer function which measures the effects of test cases in view of the user stories that represent the customer's needs and values. $\boldsymbol{x}$ is the vector which describes the yet to be determined, initially unknown, importance of the test cases. Clearly, $\boldsymbol{x}$ depends on the matrix $\boldsymbol{A}$ and the goal profile $\boldsymbol{y}$

However, since test cases are what we are looking for, the function $\boldsymbol{A}$ is not given, either. It depends on the test cases – preexisting and generated – that we use in our test suite. In practice, we start with a rule set – *Test Cases* – that can be grouped in *Test Stories* and extended as needed; see (Fehlmann, 2020). Test stories and *User Stories* allow representing the function $\boldsymbol{A}$ as a matrix. The user stories represent the requirements, based on customer needs.

Figure 2: Equation $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$ as a Matrix between User Stories and Test Stories



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

The goal vector $\boldsymbol{y}$ is labelled *Goal Test Coverage* in Figure 2; the *Achieved Result* is the product of matrix $\boldsymbol{A}$ and vector $\boldsymbol{x}$, at the bottom of the matrix. The vector $\boldsymbol{x}$ is widely unknown at the time when tests are designed. Not even its dimension – the number of tests – is obvious, nor the topics that merit being tested. Thus, there are many ways of designing a valid test strategy; however, the convergence gap (see equation (14) below) must remain small.

What Number to Put into the Matrix Cells?

The number in the matrix cells represent the total test size that correlated between the respective user story and test story. This *Cell Test Size* is the number of data movements within all test cases in the specific Test Story that pertain to the respective user story. The main problem is how to find a vector $\boldsymbol{y}$ representing qualitative or quantitative user needs, as a profile. A profile is a vector in some space of user needs with Euclidian length $= 1$. Agile teams have a process to prioritize user stories; however, they usually do not care about representing priorities as a profile vector.

Finding $\boldsymbol{A}$ and $\boldsymbol{x}$ in equation (11) is not trivial. However, in practice, the even bigger problem is that the goal vector $\boldsymbol{y}$ is often unknown. The needs of the customer, or user, in view of testing is nothing that development teams know automatically, because it involves safety, privacy and security in addition to functionality. We need a profile for all explicit and implicit requirements.

Formally, the cell numbers are constructed as follows:

- Test stories are a collection of rule sets (test cases) that share a common purpose. Let $\boldsymbol{t} \in \boldsymbol{\mathcal{S}}$ be a test story, member of some rule set $\boldsymbol{\mathcal{S}}$.
- For every test story $\boldsymbol{t}$, there is a mapping

$$\boldsymbol{map}(\boldsymbol{t}) \in \bigcup \mathfrak{S} \tag{12}$$

  where $\bigcup \mathfrak{S}$ is the set of all data movement maps within a software program, as before.
- Furthermore, there is a choice function $\|\boldsymbol{map}(\boldsymbol{t})\|_{\boldsymbol{f}}$ identifying which data movements pertain to some specific user story $\boldsymbol{f}$ and counting them.
- For each cell, we start with a rule set of test cases $\boldsymbol{t}_{i,j} \in \boldsymbol{\mathcal{S}}_j$, where i, j are the respective cell indices of the matrix $\boldsymbol{A}$ and $\boldsymbol{\mathcal{S}}_j$is the respective test story in that matrix. then

$$\sum \left\|\boldsymbol{map}\left(\boldsymbol{t}_{i,j}\right)\right\|_{\boldsymbol{f}} \tag{13}$$

  counts for each test case how many data movements pertain to the respective user story. The summation runs over all test cases $\boldsymbol{t}_{i,j} \in \boldsymbol{\mathcal{S}}_j$ for each matrix cell with index i, j.

A data movement may appear in many test cases and pertain to more than a single user story. We count the total amount of times that a data movement is used, not the data movements as for test size.

Finding the Optimum Test Cases to be Generated

There exists a family of methods – the *Analytic Hierarchy Process* (AHP) (Saaty, 2003) and *Quality Function Deployment* (QFD), explained in the series of international standards 16355 (ISO 16355-1:2015, 2015) that allow to derive such profile vectors in a professional and repeatable manner. Examples are available in (Fehlmann, 2020).

The idea is simple: if we can focus on test cases that pertain to customer needs, we have an instrument that helps us selecting those test cases that best extend testing towards a full coverage of everything that has value for the customer. Thus, we must define a choice function that achieves this.

Kranich (Fehlmann & Kranich, 2020) has tried to find such a choice function, using algebraic methods to define a sensitivity analysis procedure for any linear matrix with Eigenvector solutions, such as in QFD. However, the problem is complex, and possibly unsolvable. There may be approximation methods that can be used in practice.

The goal is to find a vector $\boldsymbol{x}$ and a matrix $\boldsymbol{A}$ such that

$$\|\boldsymbol{y} - \boldsymbol{A}\boldsymbol{x}\| < \boldsymbol{\varepsilon} \tag{14}$$

where $\|\dots\|$ denotes the Euclidean norm for vectors, called the *Convergence Gap,* and $\boldsymbol{\varepsilon}$ is the upper limit for an acceptable convergence gap. This has to do with the axiom of choice AC for the existence of real, irregular numbers in $\mathbb{R}$. Finding vector $\boldsymbol{x}$ and a matrix $\boldsymbol{A}$ is an iterative process. Finding test cases becomes equivalent to proving that a certain sequence of real numbers converges. Thus, testing is a model of combinatorial algebra.

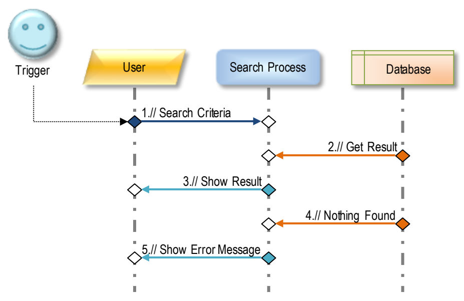
Using the combination rule (10), it is possible to generate the set of all sensible test cases. Together with the convergence gap as a metric, or hash function, the formula (14) allows to select those test cases that are relevant, and therefore limit the growth rate for newly generated test cases.

The Internet of Things (IoT) as a Simple Model

Since translating the theoretical background in practice is probably not so easy, we mention a short, simple example from IoT. The problem of generating new test cases is made considerably easier by assuming that adding another IoT "thing" adds more data movements that need to be included in tests.

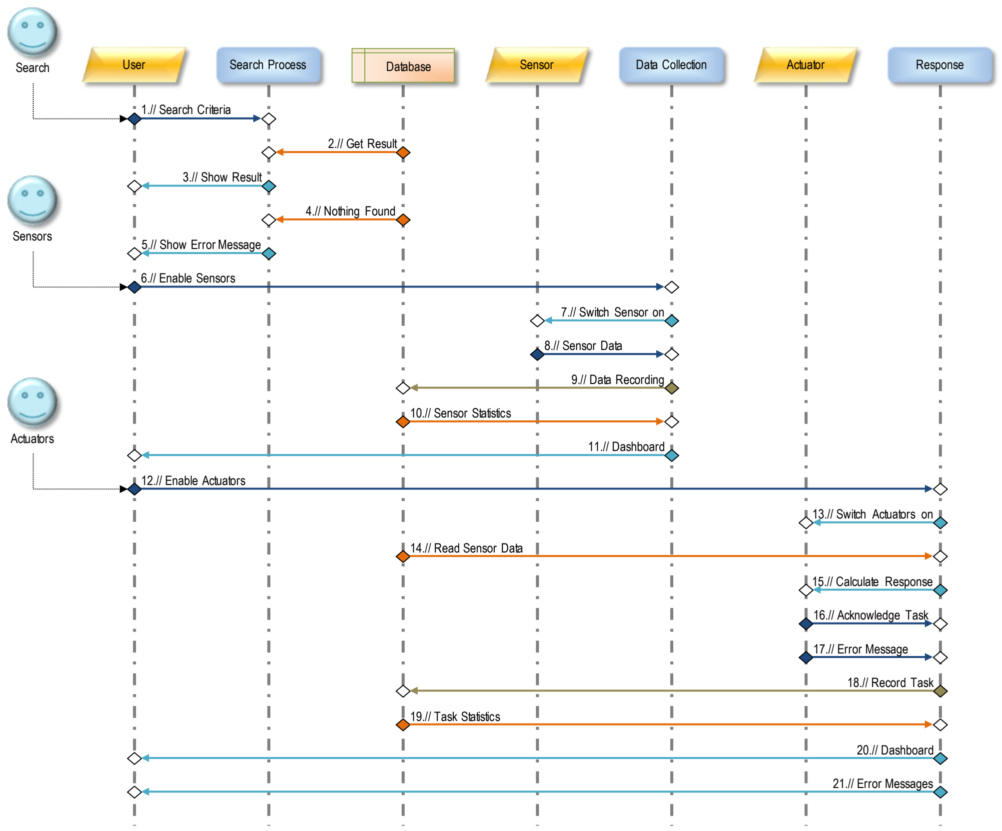
The example is a simple search app that looks for items in a database:

Figure 3: Simple Search App



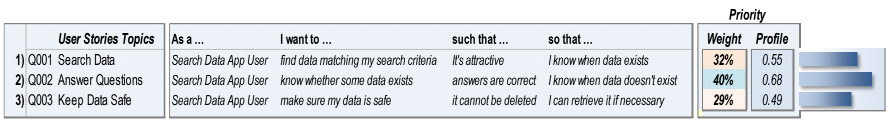
The functional size of this app is the number of data movements between objects of interest, according COSMIC (COSMIC Measurement Practices Committee, 2017). Now we add IoT devices – e.g., a sensor and an actuator that interact with the environment:

Figure 4: IoT Search Concert App



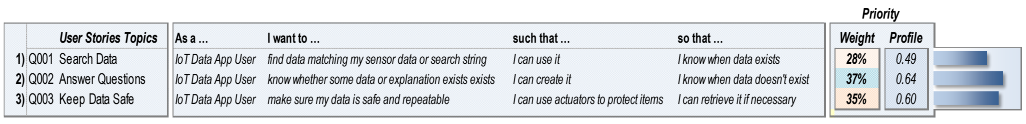
The IoT search concert still focuses on search, despite the additional functionalities. Thus, user needs for the two programs are identical. Consequently, the user stories are similar, although the user story priority profiles differ:

Figure 5: User Stories for Simple Search



	User Stories Topics	As a ...	I want to ...	such that ...	so that ...	Priority Weight	Priority Profile
1)	Q001 Search Data	*Search Data App User*	*find data matching my search criteria*	*It's attractive*	*I know when data exists*	*32%*	*0.55*
2)	Q002 Answer Questions	*Search Data App User*	*know whether some data exists*	*answers are correct*	*I know when data doesn't exist*	*40%*	*0.68*
3)	Q003 Keep Data Safe	*Search Data App User*	*make sure my data is safe*	*it cannot be deleted*	*I can retrieve it if necessary*	*29%*	*0.49*

Figure 6: User Stories for IoT Search Concert



	User Stories Topics	As a ...	I want to ...	such that ...	so that ...	Priority Weight	Priority Profile
1)	Q001 Search Data	*IoT Data App User*	*find data matching my sensor data or search string*	*I can use it*	*I know when data exists*	*28%*	*0.49*
2)	Q002 Answer Questions	*IoT Data App User*	*know whether some data or explanation exists exists*	*I can create it*	*I know when data doesn't exist*	*37%*	*0.64*
3)	Q003 Keep Data Safe	*IoT Data App User*	*make sure my data is safe and repeatable*	*I can use actuators to protect items*	*I can retrieve it if necessary*	*35%*	*0.60*

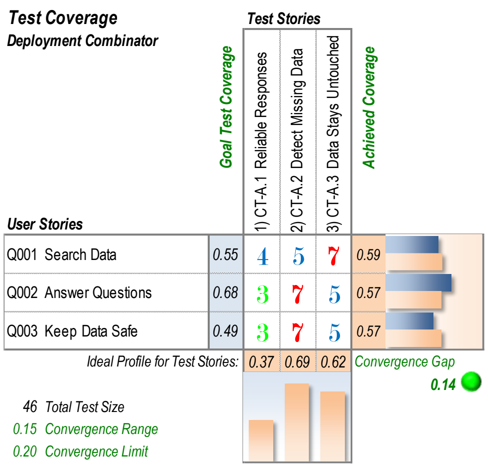
As before, the choice functions $\|\boldsymbol{map}(\boldsymbol{t})\|_f$ define which data movements pertain to which user story. This allows constructing test coverage matrices for both, the Simple Search, and the IoT Search Concert. The test stories remain the same for both apps; the test cases for Simple Search also apply for IoT Search Concert.

The IoT Search Concert needs considerably more test cases, to cover additional sensor and actuator functionality. The growth in test size (= total number of data movements in test cases according COSMIC, see above) is considerably. Test size increases from 46 to 235.

These test cases can easily be constructed by concatenating unit tests for the IoT devices with the test cases already in place for the Simple Search App, using equation (10).

Obviously, the number of possible combinations grows exponentially and would soon exceed all available test capacity. This growth can be kept under control by selecting only those new test cases that keep the convergence gap of the test coverage matrix small enough, solving $\boldsymbol{y} = \boldsymbol{Ax}$:

Figure 7: Simple Search Test Coverage

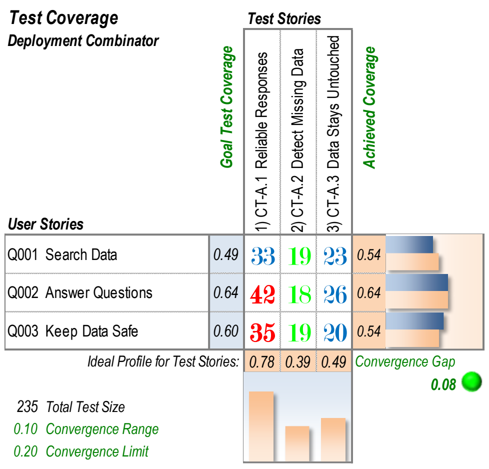


Test Coverage
Deployment Combinator

User Stories	Goal Test Coverage	1) CT-A.1 Reliable Responses	2) CT-A.2 Detect Missing Data	3) CT-A.3 Data Stays Untouched	Achieved Coverage
Q001 Search Data	0.55	4	5	7	0.59
Q002 Answer Questions	0.68	3	7	5	0.57
Q003 Keep Data Safe	0.49	3	7	5	0.57
Ideal Profile for Test Stories:		0.37	0.69	0.62	Convergence Gap 0.14 ●

46 Total Test Size
0.15 Convergence Range
0.20 Convergence Limit

Figure 8: IoT Search Test Coverage



Test Coverage
Deployment Combinator

User Stories	Goal Test Coverage	1) CT-A.1 Reliable Responses	2) CT-A.2 Detect Missing Data	3) CT-A.3 Data Stays Untouched	Achieved Coverage
Q001 Search Data	0.49	33	19	23	0.54
Q002 Answer Questions	0.64	42	18	26	0.64
Q003 Keep Data Safe	0.60	35	19	20	0.54
Ideal Profile for Test Stories:		0.78	0.39	0.49	Convergence Gap 0.08 ●

235 Total Test Size
0.10 Convergence Range
0.20 Convergence Limit

For *Artificial Intelligence* (AI), such search algorithms are typical methods. Using the convergence gap as a hash function for selecting meaningful test cases limits the growth of the search tree for test cases. For more details, see (Fehlmann, 2020).

The Axiom of Choice and Artificial Intelligence (AI)

The misconception about computability of non-measurable structures – such as real numbers ℝ – is also responsible for a very actual problem: some people believe that AI can solve problems; however, AI always approximates solving a problem. If AI comes without anything resembling the convergence gap it is most probably useless. You cannot rely on AI decisions without measuring accuracy.

Nevertheless, testing AI is possible, and not too difficult (Fehlmann & Kranich, 2019). However, it cannot test the neuronal network – e.g., the *Support Vector Machine* (Gunn, 1998) – itself but only its behavior in certain test situations. This is also explaining intuitively why testing never can prove anything. Testing software is always an approximation by a finite number of test cases, be it AI or traditional algorithmic programming. We always need a constructive choice function that selects relevant test cases for the approximation.

This suggests also that testing without a convergence gap is deceptive and potentially misleading.

Testing AI (Fehlmann & Kranich, 2019) is done by construction of data movement maps that describe the expected behavior. These data movement maps do not represent the program code; rather the behavior expected by the user.

The choice function selecting relevant test cases is therefore relevant. It is all but obvious which to choose, but it makes testing AI a matter of understanding its expectations in an intelligent machine. To believe that AI is intelligent by itself is like believing that the Banach-Tarski sphere can be split into two.

Open Questions

Besides sensitivity analysis for matrices, there is one very stringent question open: can we define combinators that help us in generating meaningful additional tests? Like what Engeler did for neural networks?

Furthermore, is there a connection between sensitivity analysis and such combinators? Both questions may not only lead to practical solutions, but also interesting theoretical insight in the role of the axiom of choice for software engineering.

Conclusions

Computer science uses choice functions only in a constructive way; existence of a choice always means existence of an algorithm that does the choice. This is counter-intuitive to human perception of the world but reflect the standpoint of mathematical logic.

Arrow terms are an extremely rich structure for representing quite different structures such as the way how the brain thinks, the way how to focus on customer needs by *Quality Function Deployment* (QFD), see (Fehlmann, 2002), and testing of complex, software-intense systems with thousands of *Embedded Control Units* (ECU).

Acknowledgement. The authors would like to thank the anonymous referees for their valuable comments and suggestions which led to an improvement of the paper.

References

Banach, S. & Tarski, A., 1924. Sur la décomposition des ensembles de points en parties respectivement congruentes. *Fundamenta Mathematicae,* Volume 6, p. 244–277.

Barendregt, H. P., 1977. The Type-Free Lambda-Calculus. Dans: J. Barwise, éd. *Handbook of Math. Logic.* Amsterdam: North Holland, pp. 1091 -1132.

Barwise, J. et al., 1977. *Handbook of Mathematical Logic.* Studies in Logic and the Foundations of Mathematics éd. Amsterdam, NL: North-Holland Publishing Company.

Bell, D., 2004. *UML basics: The Sequence Diagram – Introductory Level,* Armonk, NY: IBM DeveloperWorks.

COSMIC Measurement Practices Committee, 2017. *The COSMIC Functional Size Measurement Method – Version 4.0.2 – Measurement Manual,* Montréal: The COSMIC Consortium.

Curry, H. & Feys, R., 1958. *Combinatory Logic, Vol. I.* Amsterdam: North-Holland.

Curry, H., Hindley, J. & Seldin, J., 1972. *Combinatory Logic, Vol. II.* Amsterdam: North-Holland.

Engeler, E., 1981. Algebras and Combinators. *Algebra Universalis,* pp. 389-392.

Engeler, E., 1995. *The Combinatory Programme.* Basel, Switzerland: Birkhäuser.

Engeler, E., 2019. Neural algebra on "how does the brain think?". *Theoretical Computer Science,* Volume 777, pp. 296-307.

Fehlmann, T. M., 1981. *Theorie und Anwendung des Graphmodells der Kombinatorischen Logik,* Zürich, CH: ETH Dissertation 3140-01.

Fehlmann, T. M., 2002. *QFD as Algebra of Combinators.* Tokyo, Japan, 8th International QFD Symposium, ISQFD 2002.

Fehlmann, T. M., 2016. *Managing Complexity - Uncover the Mysteries with Six Sigma Transfer Functions.* Berlin, Germany: Logos Press.

Fehlmann, T. M., 2020. *Autonomous Real-time Testing - Testing Artificial Intelligence and Other Complex Systems.* Berlin, Germany: Logos Press.

Fehlmann, T. M. & Kranich, E., 2019. Testing Artificial Intelligence by Customers' Needs. *Athens Journal of Sciences,* 6(4), pp. 265-286.

Fehlmann, T. M. & Kranich, E., 2020. *A Sensitivity Analysis Procedure for QFD,* Duisburg: to appear.

Gunn, S., 1998. *Support Vector Machines for Classification and Regression,* Southampton: ISIS Technical Report, University of Southampton.

ISO 16355-1:2015, 2015. *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,* 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.

Pawlikowski, J., 1991. The Hahn–Banach theorem implies the Banach–Tarski paradox. *Fundamenta Mathematicae,* Volume 138, p. 21–22.

Potter, M. D., 2004. *Set Theory and Its Philosophy.* Oxford, UK: Oxford University Press.

Rauglaudre, D. d., 2017. Formal Proof of Banach-Tarski Paradox. *Journal of Formalized Reasoning,* 10(1), pp. 37-49.

Robinson, R. M., 1947. On the Decomposition of Spheres. *Fundamenta Mathematicae,* Volume 34, pp. 246-260.

Saaty, T. L., 2003. Decision-making with the AHP: Why is the principal eigenvector necessary?. *European Journal of Operational Research,* Volume 145, pp. 85-91.

Tao, T., 2011. *An Introduction to Measure Theory.* Los Angeles, CA: WordPress.com.

Zachos, E., 1978. *Kombinatorische Logik und S-Terme,* Zurich: ETH Dissertation 6214.