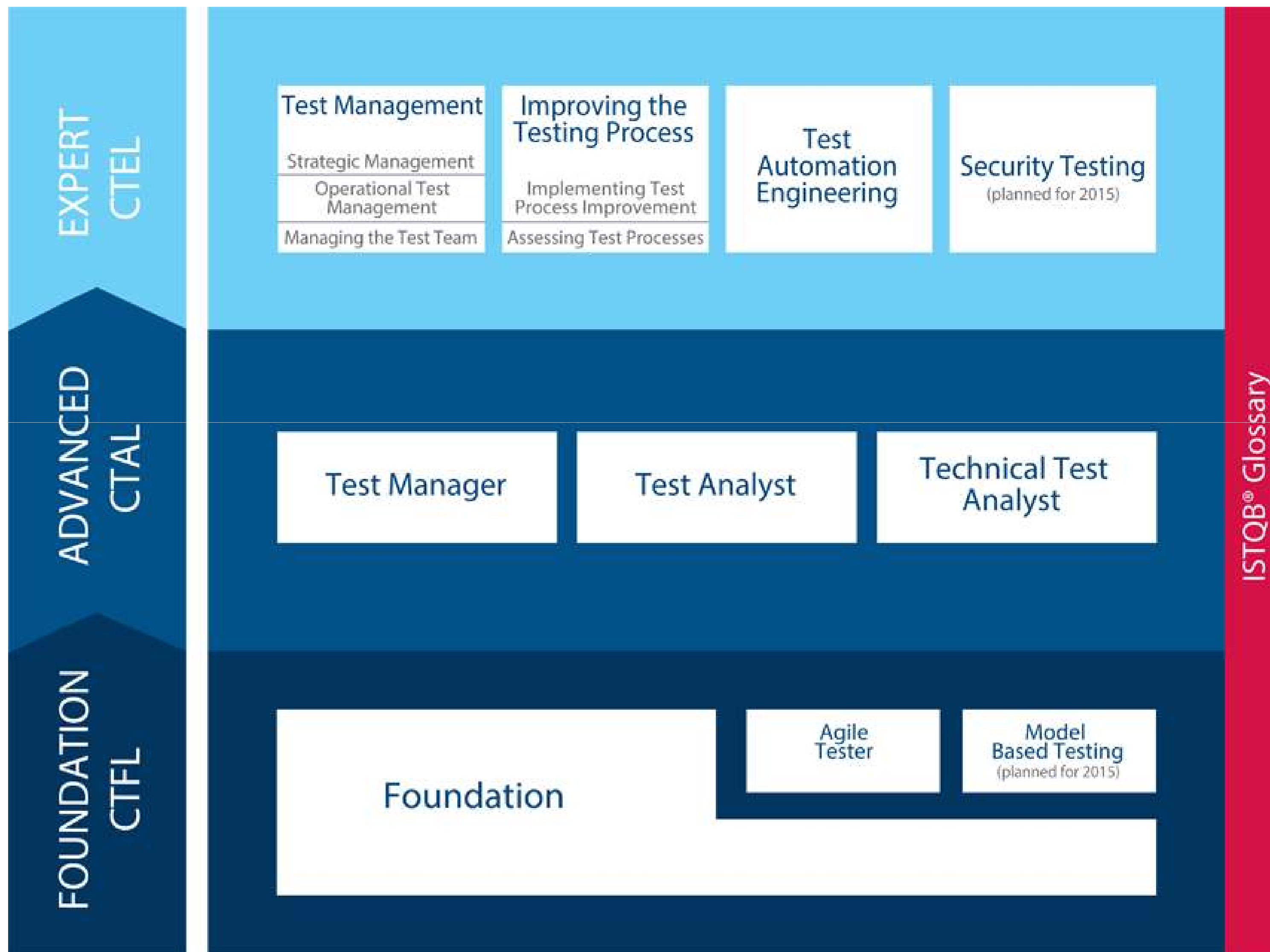
Tesztelési ág

VIMIMA11 Rendszertervezés és –integráció

Scherer Balázs

ISTQB: *International Software Testing Qualifications Board*



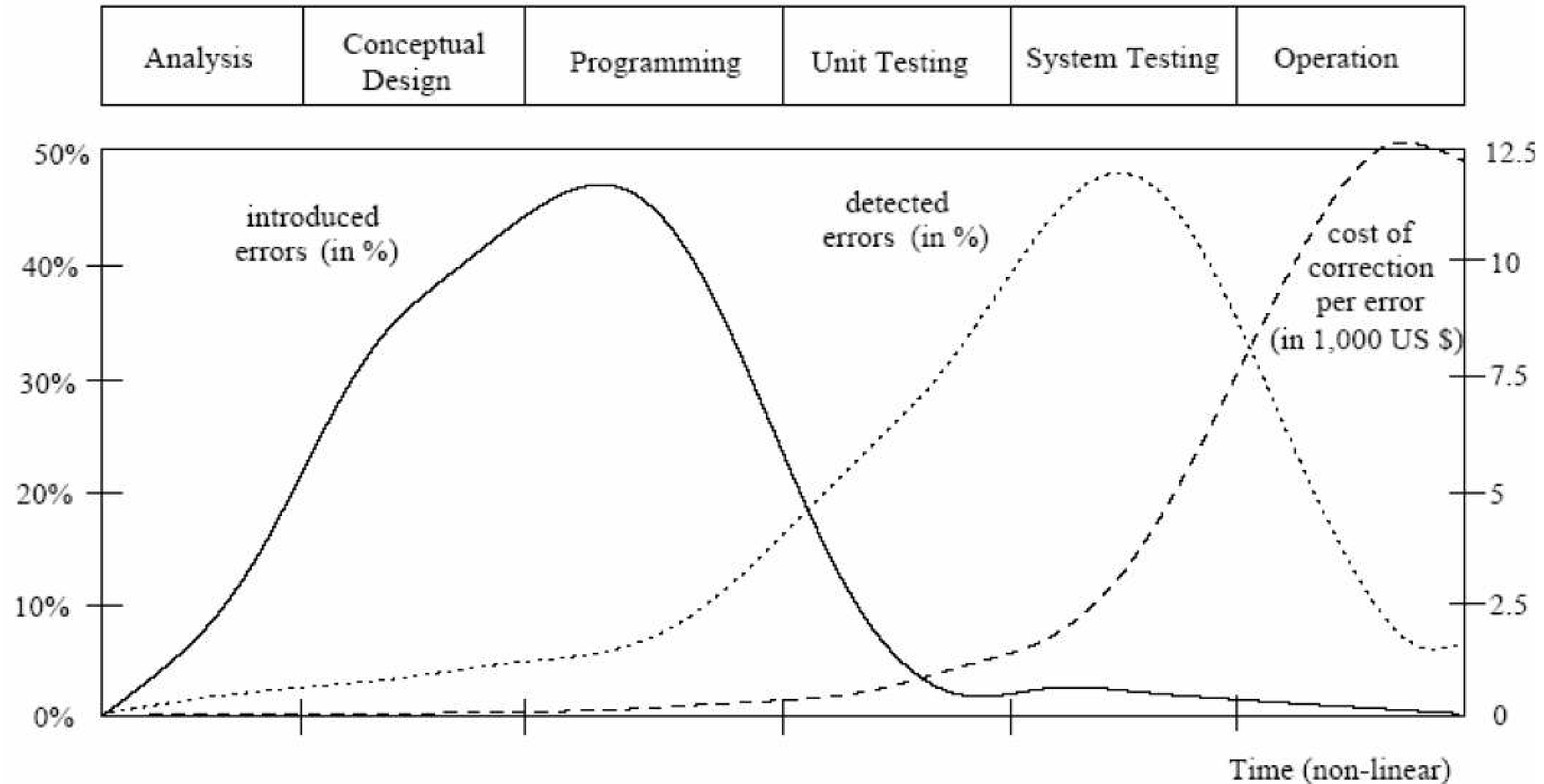
Szoftverhiba és keletkezése

- ***Error:*** Emberi tevékenység, ami nem kívánt eredményre vezet
- ***Fault:*** Az ***Error*** megjelenése a kódban, más néven ***bug***. Ennek végrehajtása ***Failure***-höz vezethet
- ***Failure:*** A szoftver elvártól eltérő viselkedése, tehát megtalált hiba (defects)

Tesztelési alaptételek

- A tesztelés a hibák jelenlétét mutatja meg
  - o De nem tudja azt garantálni, nincs hiba a rendszerben. A tesztelés csökkenti a nem észrevett hibák valószínűségét, de az, hogy egy tesztelés során nem találtunk hibát még nem bizonyítja, hogy a kód hibamentes.
- Kimerítő tesztelés nem lehetséges
  - o Minden bemeneti kombinációt letesztelni triviális esetektől eltekintve lehetetlen. A kimerítő tesztelés helyett az értékel kockázatok és prioritások alapján fókuszáljuk a tesztelési erőforrásokat
- A tesztelést minél korábbi fázisban el kell kezdeni

Miért fontos a korai és szisztematikus tesztelés?

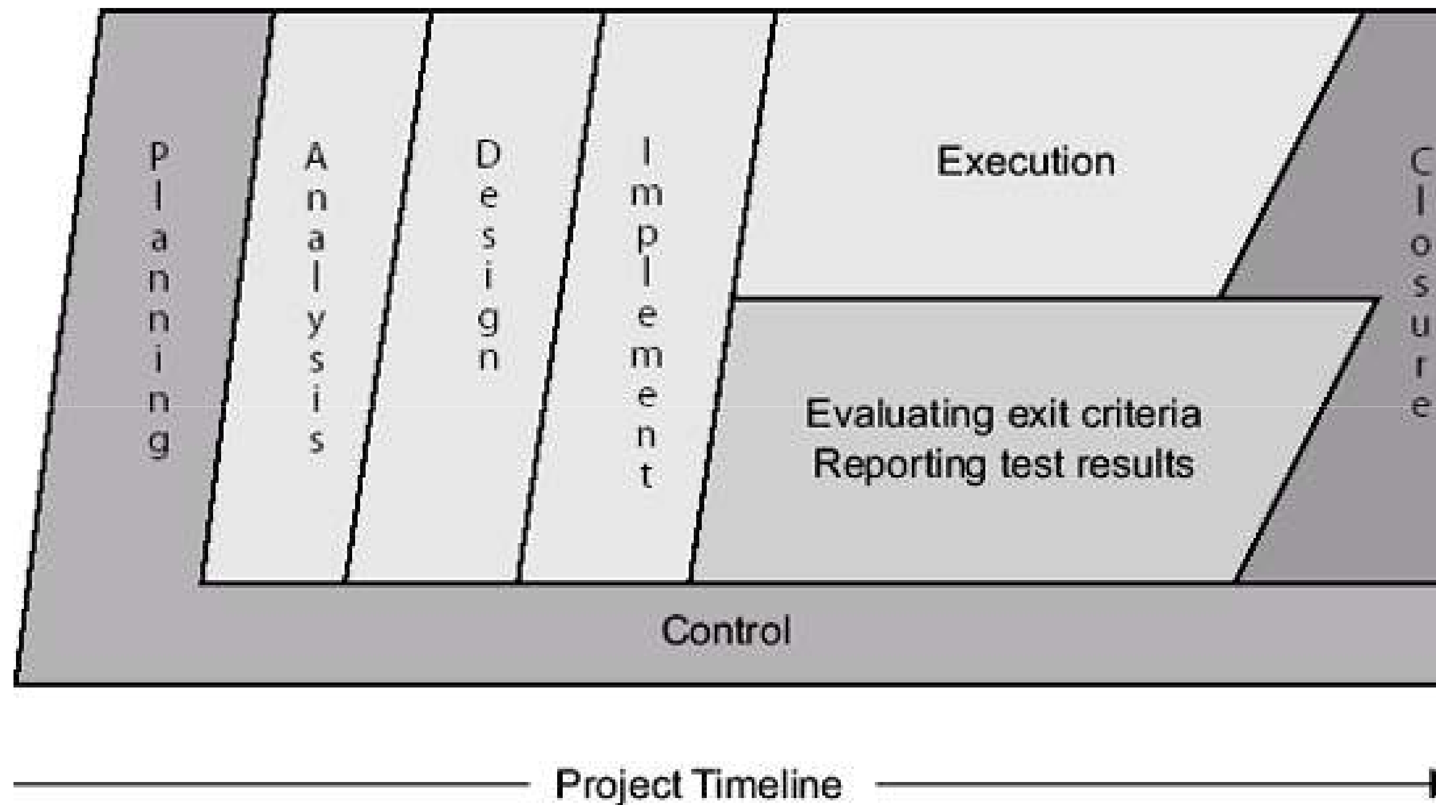


From: P. Liggesmeyer et al., Qualitätssicherung Software-basierter technischer Systeme, Informatik Spektrum, 21:249-258, 1998. Quoted after J.P. Katoen, Principles of Model Checking, 2004/5. Copyright © by the authors.

Tesztelési alaptételek *folytatás*

- Hibaklaszterezés
  - A legtöbb esetben a hibák nagy részét a rendszer viszonylag ki mennyiségű, de komplex része tartalmazza
- Növényvédőszer paradoxon
  - Ha ugyanazokat a teszteket futtatjuk folyamatosan, akkor a rendszer „immunissá” válik velük szemben, és egy idő után nem találunk új hibákat. Tehát a teszt eseteket időről időre felül kell vizsgálni
- A tesztelés környezet függő
  - Más követelmények vannak egy safety critical és egy PC szoftver tesztelésénél
- A hibamentesség önmagában nem elegendő
  - A hibamentes tesztek után is lehet a rendszer a felhasználó számára használhatatlan

Tesztelési életciklus



Teszt tervezés és felügyelés

- A teszt hatáskörének és a kockázatoknak az elemzése a teszt céljának meghatározása
  - Mit tesztelünk, szoftver komponenst, rendszer komponenst, teljes rendszert?
  - Milyen technikai kockázatai vannak a tesztelt eszköznek és projectnek
  - Mi a teszt célja? Hibák kimutatása, a specifikációnak való megfelelés, annak a demonstrálása, hogy a rendszer valamilyen célra megfelelő?
- A tesztelési megközelítés meghatározása
  - Milyen technológiával és erőforrásokkal kell tesztelnünk?
  - Milyen fedettséget kell meghatároznunk?

Teszt tervezés és felügyelés *folytatás*

- A teszt stratégia létrehozása
- A szükséges erőforrások meghatározása
- A teszteléshez szükséges feladatok meghatározása

- Az exit kritérium meghatározása
  - A tesztelés nem kifulladásig tart, vagy ameddig van rá idő. Szükséges egy olyan kritérium, például fedettség mérték, ami teljesülése esetén a tesztelést befejezettnek tekintjük.

- Felügyeleti feladatok
  - Tesztlés eredményeinek analizálása, a folyamat haladásának ellenőrzése. A szükséges módosítások végrehajtása

Teszt analízis és tervezés

- A teszt alapjainak áttekintése
  - Termék kockázatok, tervek, specifikációk, interfész leírások
- A teszt feltételek (**test condition**) azonosítása
  - Általános áttekintése annak, hogy mit is szeretnénk tesztelni
- Tesztek tervezése
  - A teszt feltételek alapján a megfelelő tesztelési eljárások kiválasztása
- A rendszer követelményeinek tesztelhetőség vizsgálata
  - Egyértelműek és jól meghatározottak-e a követelmények? Lehet-e hozzájuk tesztet esetet készíteni?
- A szükséges teszt környezet megtervezése

Teszt implementáció és végrehajtás

- A teszt feltételekből teszt eseteket csinálunk és végrehajtjuk ezeket

- Teszt esetek (**test case**) kifejlesztése és priorizálása
- Teszt készlet (**test suite**) készítése a teszt esetekből
- Tesztek végrehajtása
- A teszt log ellenőrzése
- Az elvárt eredménytől eltérő eredmények reportálása

Ki végezze a tesztet?

- Tudja-e a fejlesztő saját produktumát tesztelni?
  - Nem: mert elnéz problémákat, (pl. a rosszul értelmezi a követelményt)
  - Nem: mert nem tesztel mindent (pl. gyakran megfeledkezik a negatív tesztről)
  - Igen: senki sem ismeri a fejlesztőnél jobban a produktumot

- Miért van szükség független TestCenter-re
  - \+ elfogulatlan (nem “saját”hibákat talál meg)
  - \+ ismeri a tesztelés módszertanát
  - \- rendszer (termék) knowhow-val rendelkezik

A teszt vége kritérium ellenőrzése a teszt lezárása

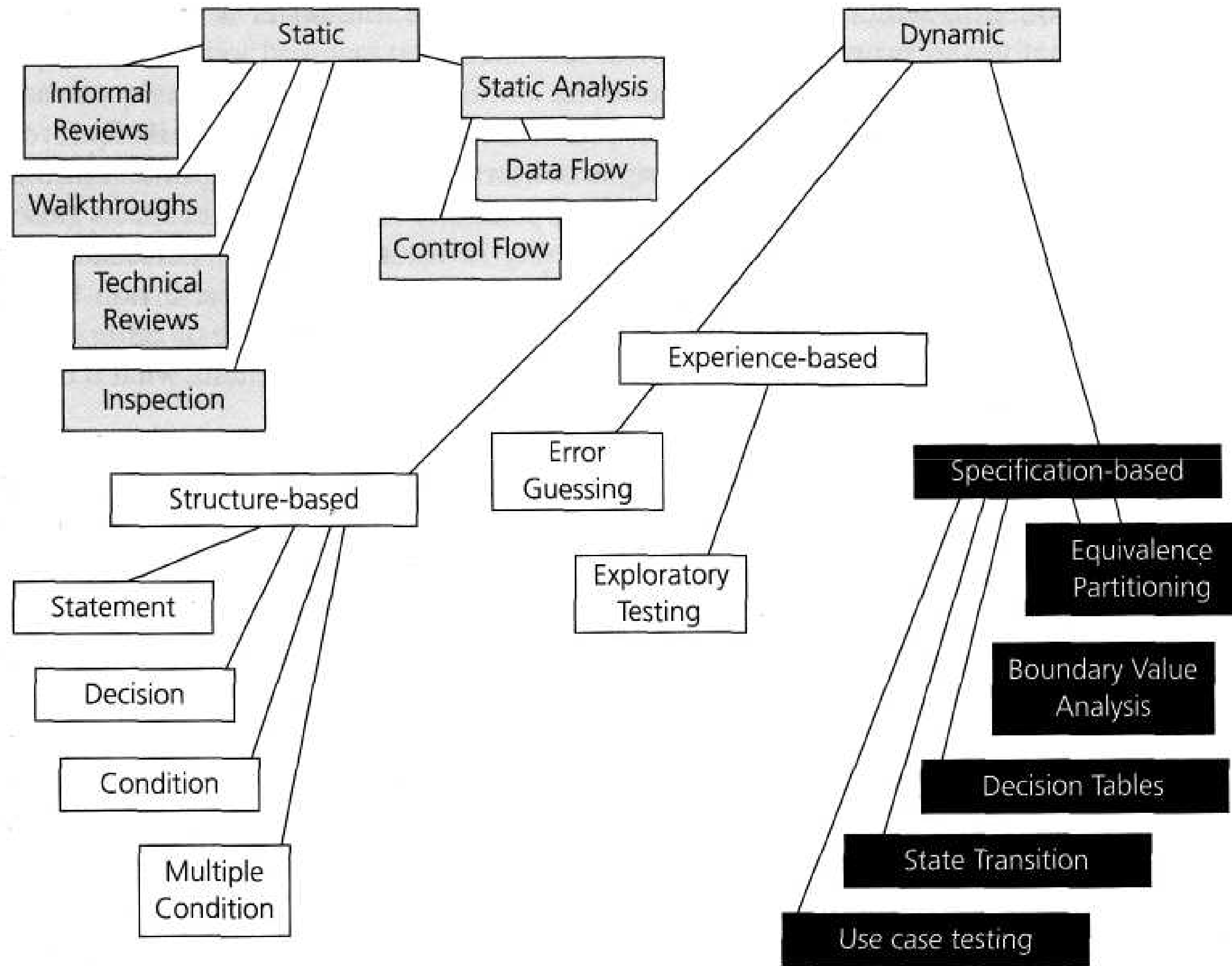
- A teszt eredményeknek az **exit criteria**-val való összevetése
- Annak jelzése, hogy több, vagy más tesztekre van még szükség
- **Test summary report** elkészítése

- A teszt környezetének archiválása

Teszt típusok

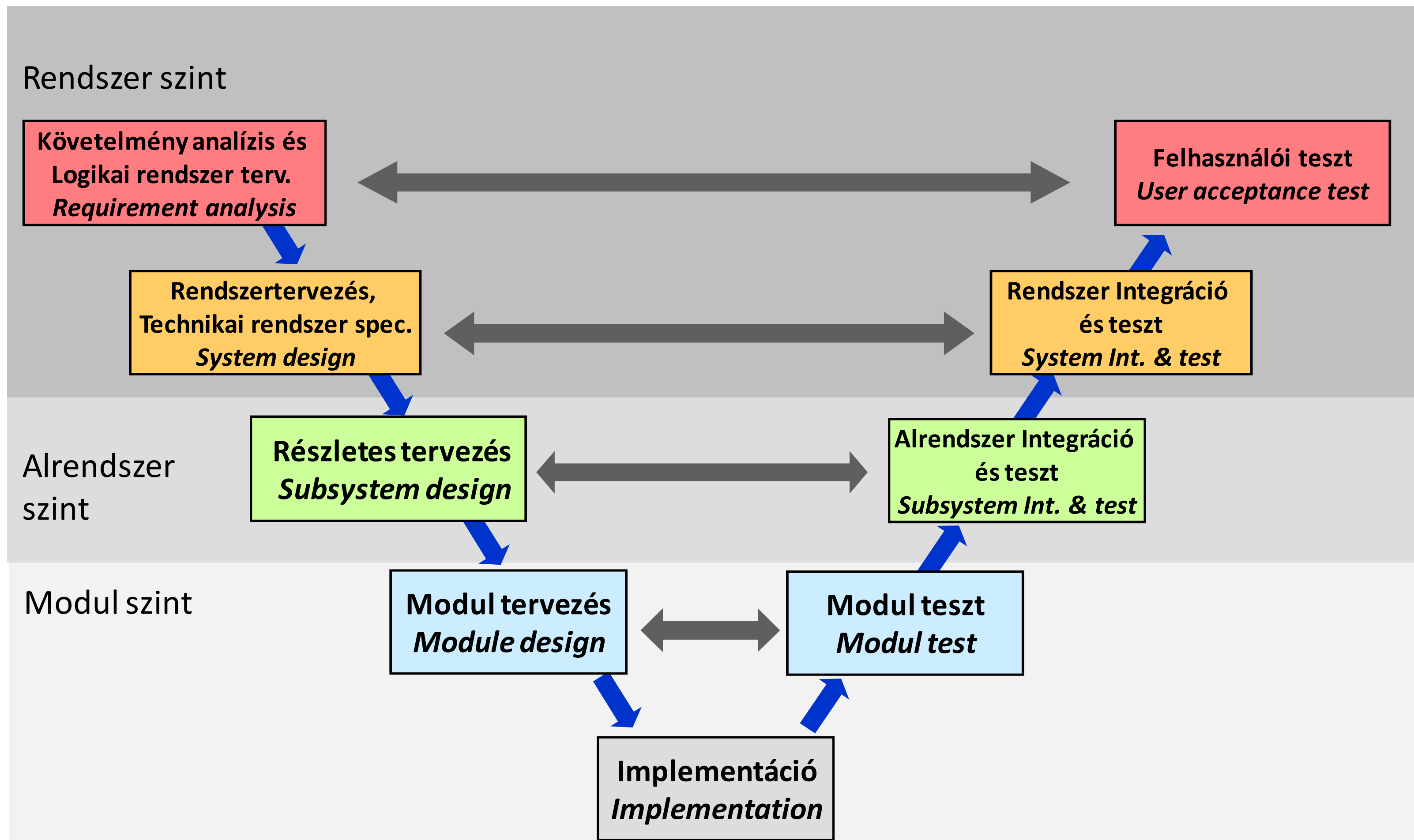
- Funkcionális tesztek
  - Általában fekete doboz tesztek
  - Specifikáció alapúak
- Nem funkcionális tesztek
  - Kezelhetőség, karbantarthatóság, megbízhatóság, hatékonyság
  - Performance, Load, Stress tesztek
- Struktúra alapú tesztek
  - Fehér doboz tesztek
  - Kódfedettséghez köthető tesztek
- Változáshoz köthető tesztek
  - Megerősítő (Confirmation) teszt: hibajavítás után annak ellenőrzése, hogy valóban megjavult-e a rendszer
  - Regressziós teszt: szoftver változtatás hatására egy szokásos teszt készlet (már egyszer helyesen végrehajtott) végrehajtása, hogy nem került-e be új hiba a rendszerbe

Teszt technológiák



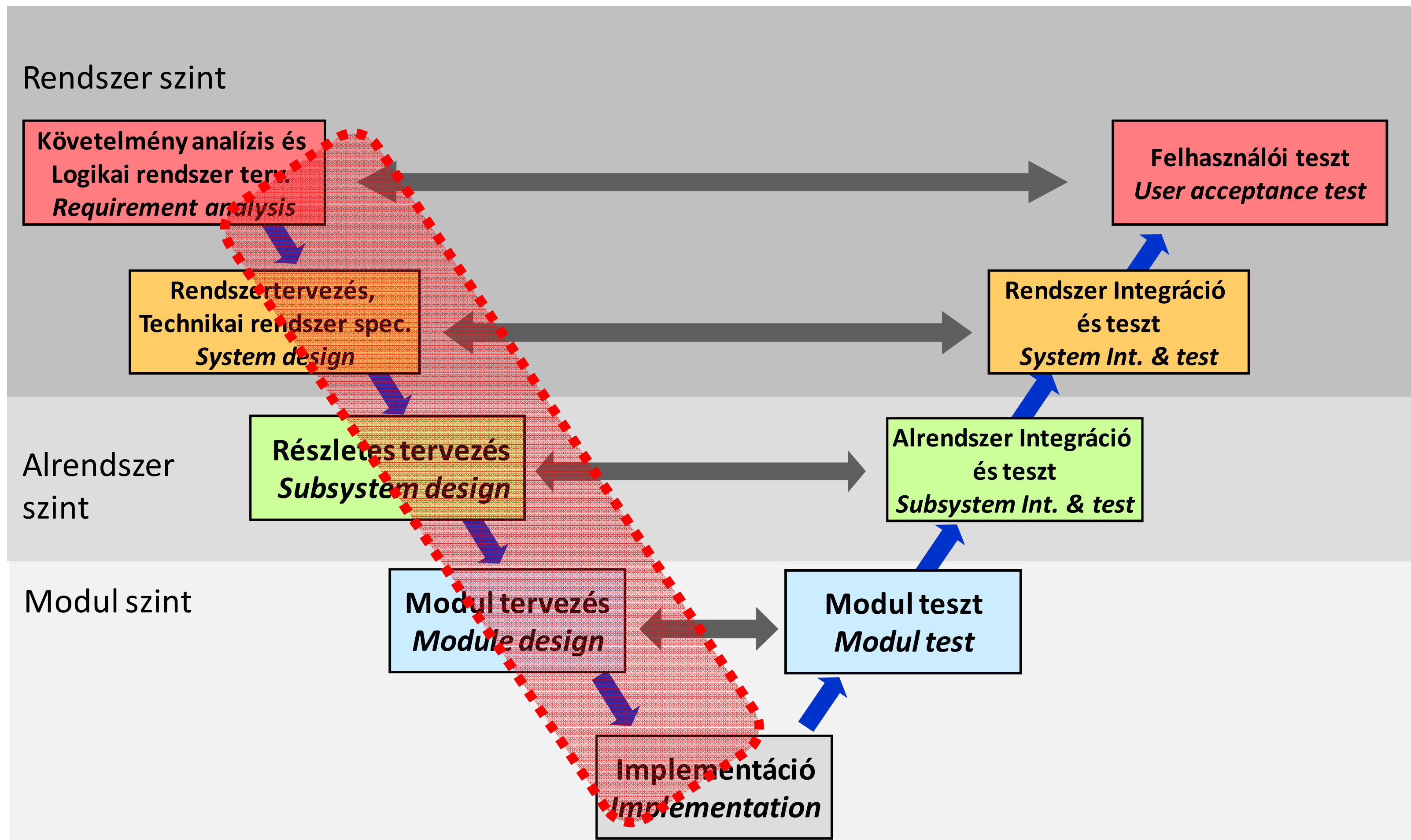
Statikus technológiák

- Kód, vagy rendszer működése nélkül végrehajthatóak
  - Korai fázisban is alkalmazhatóak

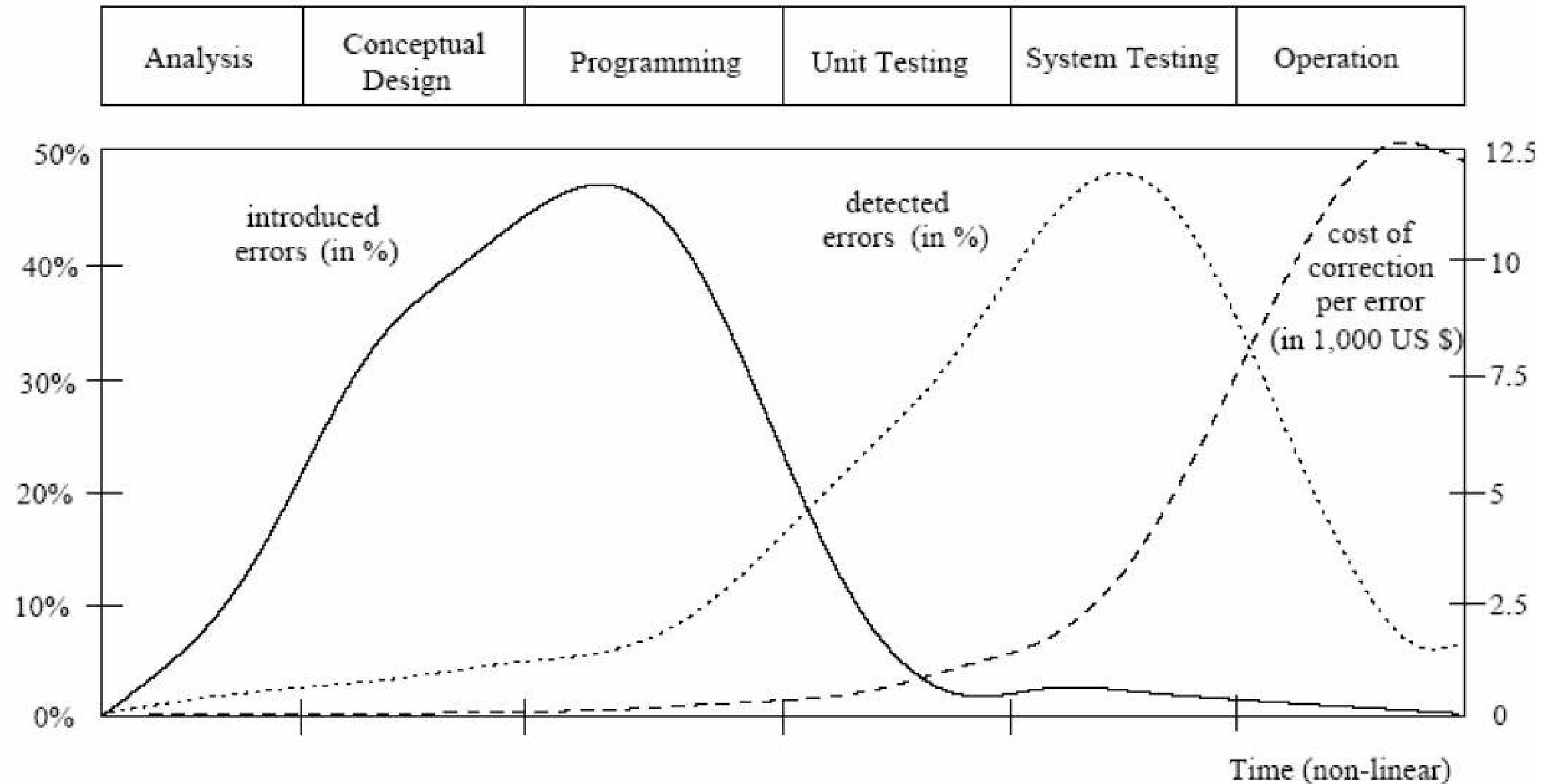


Statikus technológiák

- Kód, vagy rendszer működése nélkül végrehajthatóak
  - Korai fázisban is alkalmazhatóak



Miért fontos a korai és szisztematikus tesztelés?



From: P. Liggesmeyer et al., Qualitätssicherung Software-basierter technischer Systeme, Informatik Spektrum, 21:249-258, 1998. Quoted after J.P. Katoen, Principles of Model Checking, 2004/5. Copyright © by the authors.

Statikus teszt technikák: review-k

- Walkthrough
  - A dokumentum, vagy terv készítői bemutatja azt a többieknek
  - Cél a közös konszenzus, megértés elérése és visszacsatolás kapása
  - A meetinget a tervezők vezetik
- Technical review
  - A technikai részletek és döntések ellenőrzése
  - Tipikusan külső expertek bevonásával
  - Nem teljesen formális a dolog, de dokumentált
  - Egy moderátor, vagy expert vezeti a meetinget
- Inspections
  - A revisorok a szükséges dokumentációkat részletesen megvizsgáljak a meeting előtt szabályokra, check listákra hagyatkozva
  - A meetinget egy képzett moderátor vezeti
  - Előre meghatározott szabályozott forgatókönyv szerint zajlik a dolog
  - A megtalált hibák dokumentálásra kerülnek és issue kezelés részévé válnak

Statikus teszt technikák: statikus analízis

- Kódolási szabályoknak megfelelés ellenőrzése
  - MISRA-C kompatibilitás ellenőrzés
- Kódmetrikák
  - A kód 20%-a okozza a problémák 80%-át
  - Ciklomatikus komplexitás (Cyclomatic complexity) kiderítése
  - Komment sűrűség
- Adatfolyam analízis
  - Nem használunk-e inicializálatlan adatot
  - Nincs e túlcsordulás, alulcsordulás, 0-val való osztás
- Vezérlési folyam analízis
  - Nincs-e elérhetetlen kódrészlet

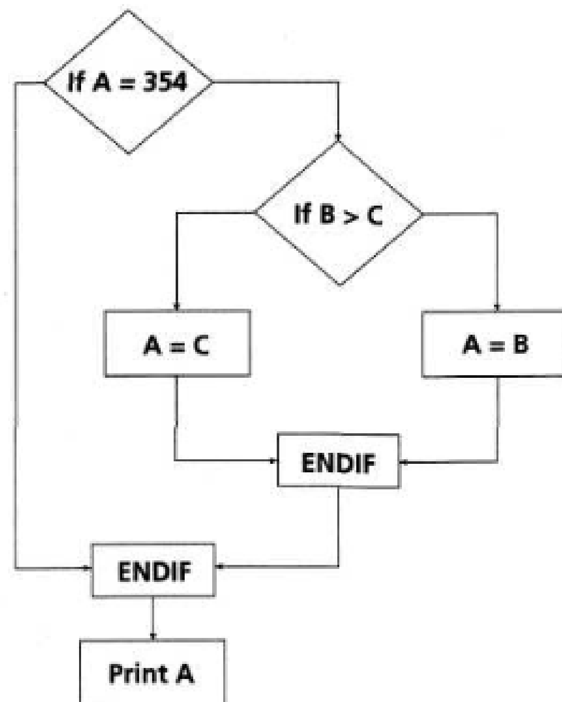
Ciklomatikus komplexitás

- A kód komplexitásáról ad információt, hasznos, mert meg lehet becsülni a tesztelés komplexitását

$$\boldsymbol{M} = E - N + 2$$

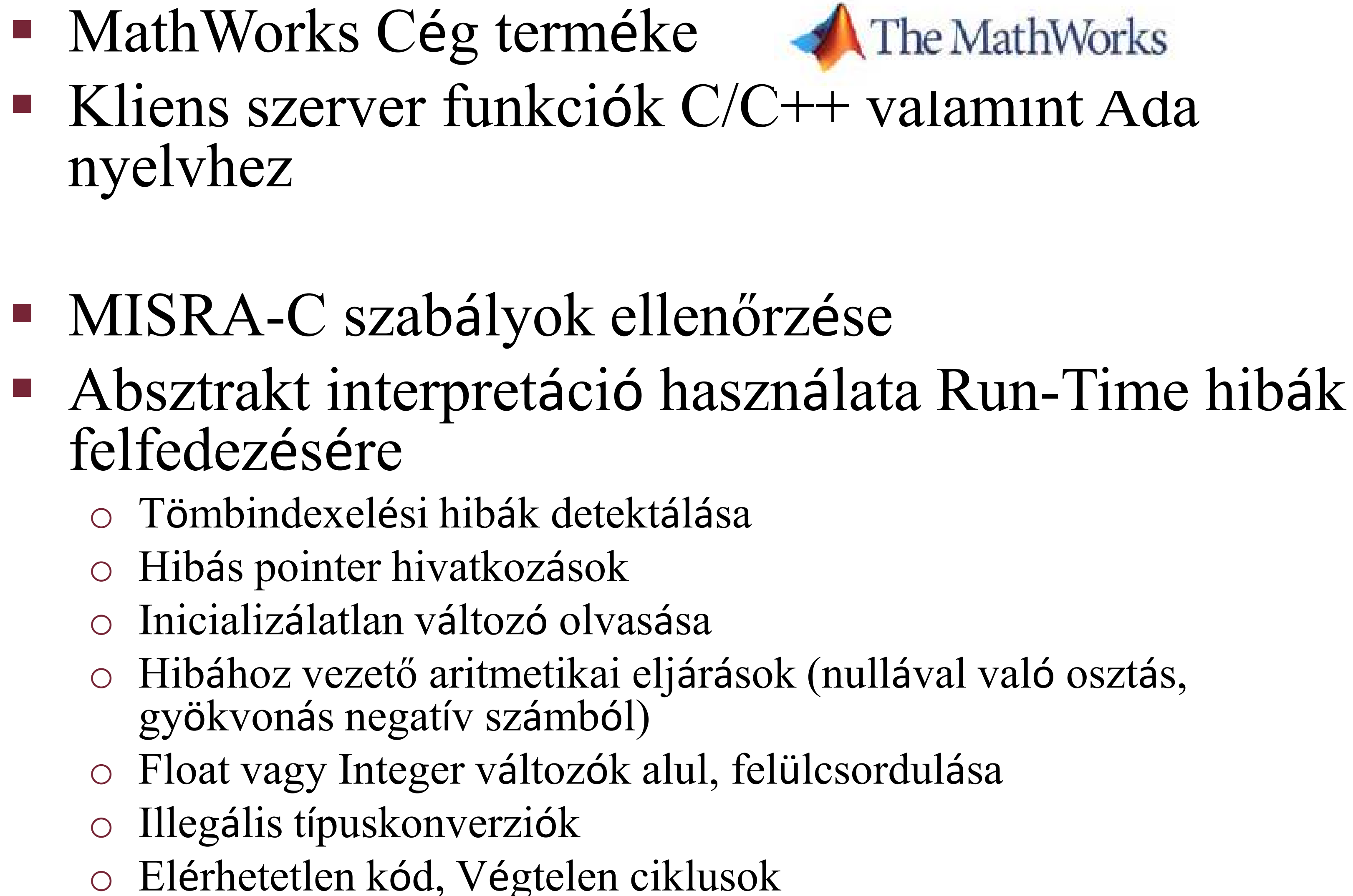
E: *A gráf éleinek száma*

N: *A gráfban lévő csúcsok száma*



8-7+2 = 3

Statikus analízis tool példa: PolySpace statikus kód analizátor



- MathWorks Cég terméke
- Kliens szerver funkciók C/C++ valamint Ada nyelvhez

- MISRA-C szabályok ellenőrzése
- Absztrakt interpretáció használata Run-Time hibák felfedezésére
  - Tömbindexelési hibák detektálása
  - Hibás pointer hivatkozások
  - Inicializálatlan változó olvasása
  - Hibához vezető aritmetikai eljárások (nullával való osztás, gyökvonás negatív számból)
  - Float vagy Integer változók alul, felülcsordulása
  - Illegális típuskonverziók
  - Elérhetetlen kód, Végtelen ciklusok

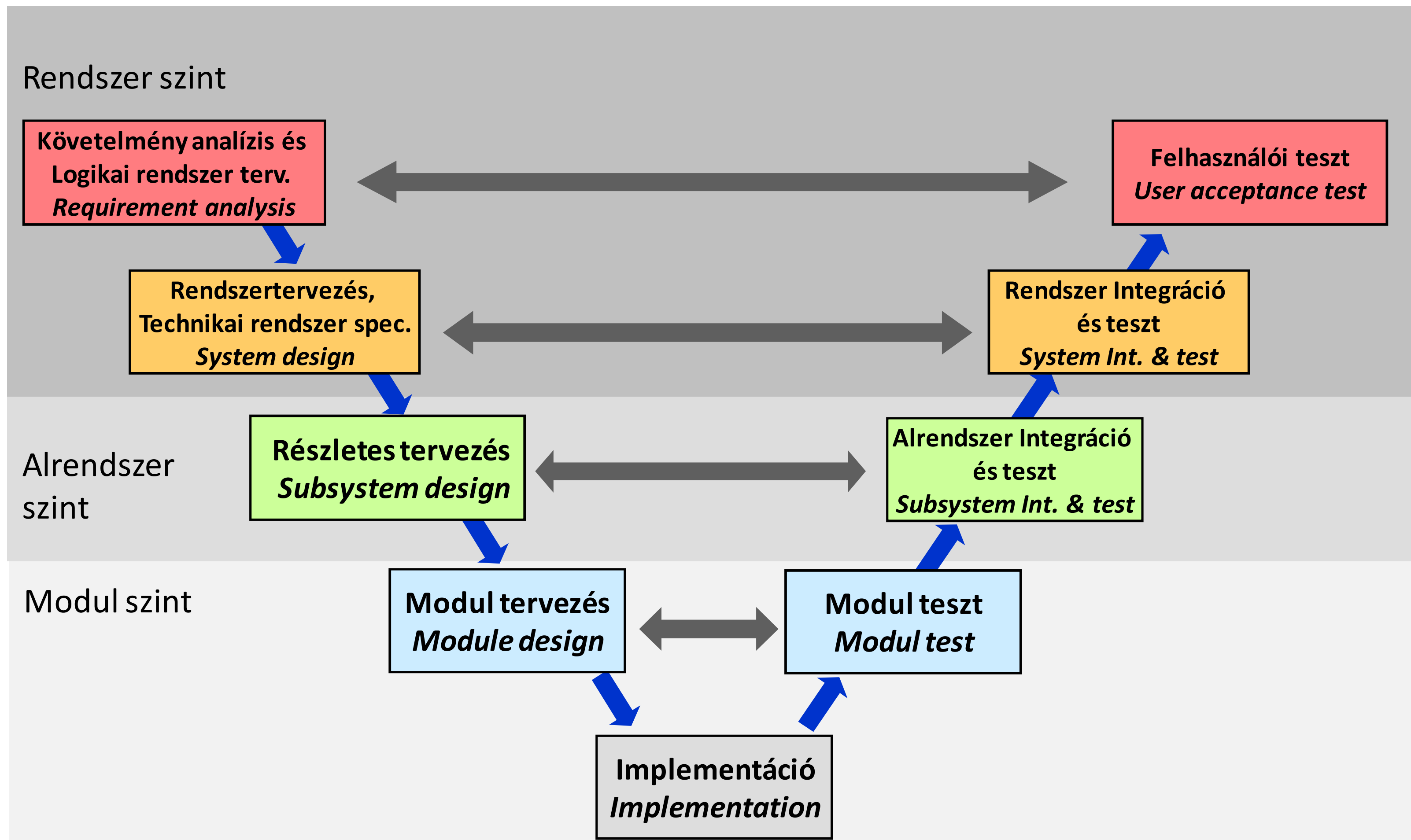
PolySpace kliens működés közben


```
static void Pointer_Arithmetic (void)
{
  int array[100];
  int i, *p = array;

  for(i = 0; i < 100; i++, p++)
    *p = 0;

  if(get_bus_status() > 0) {
      if (get_oil_pressure() > 0)
        *p = 5;
      else
        i++;
    }

  i = get_bus_status();
  if (i >= 0)  { *(p-i) = 10; }

  if ((0 < i) && (i <= 100)) {
      p = p - i;
      *p = 5;
    }
}
```

- **Zöld:**
  bizonyítottan helyes minden körülmények között
- **Piros:**
  bizonyítottan rossz minden körülmények között
- **Szürke:**
  bizonyítottan elérhetetlen kód
- **Narancs:**
  bizonyos körülmények között lehet rossz

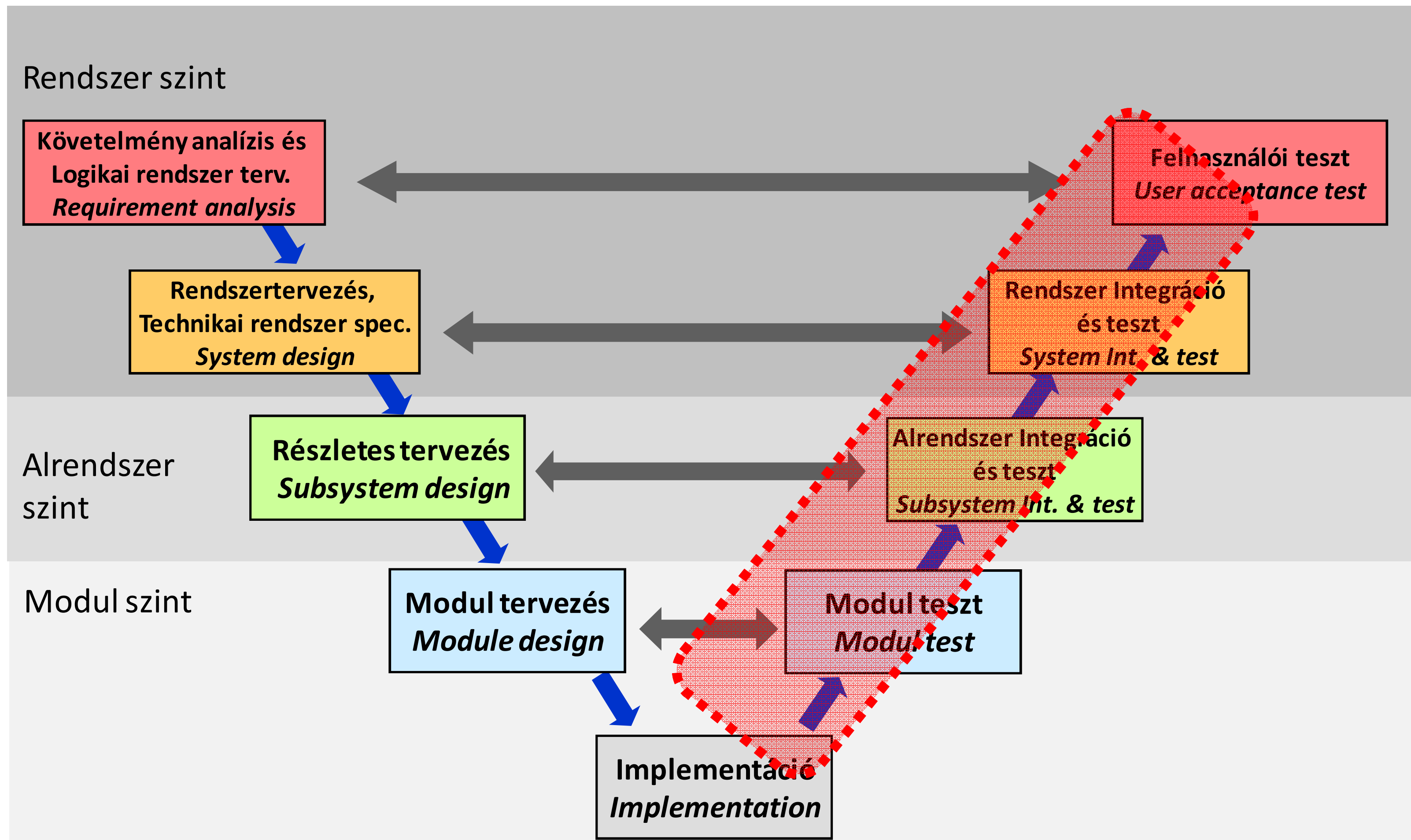
Dinamikus teszt technológiák

- Kód, vagy rendszer működése szükséges a teszteléshez
  - Korai fázisból a követelmények vehetők csak át

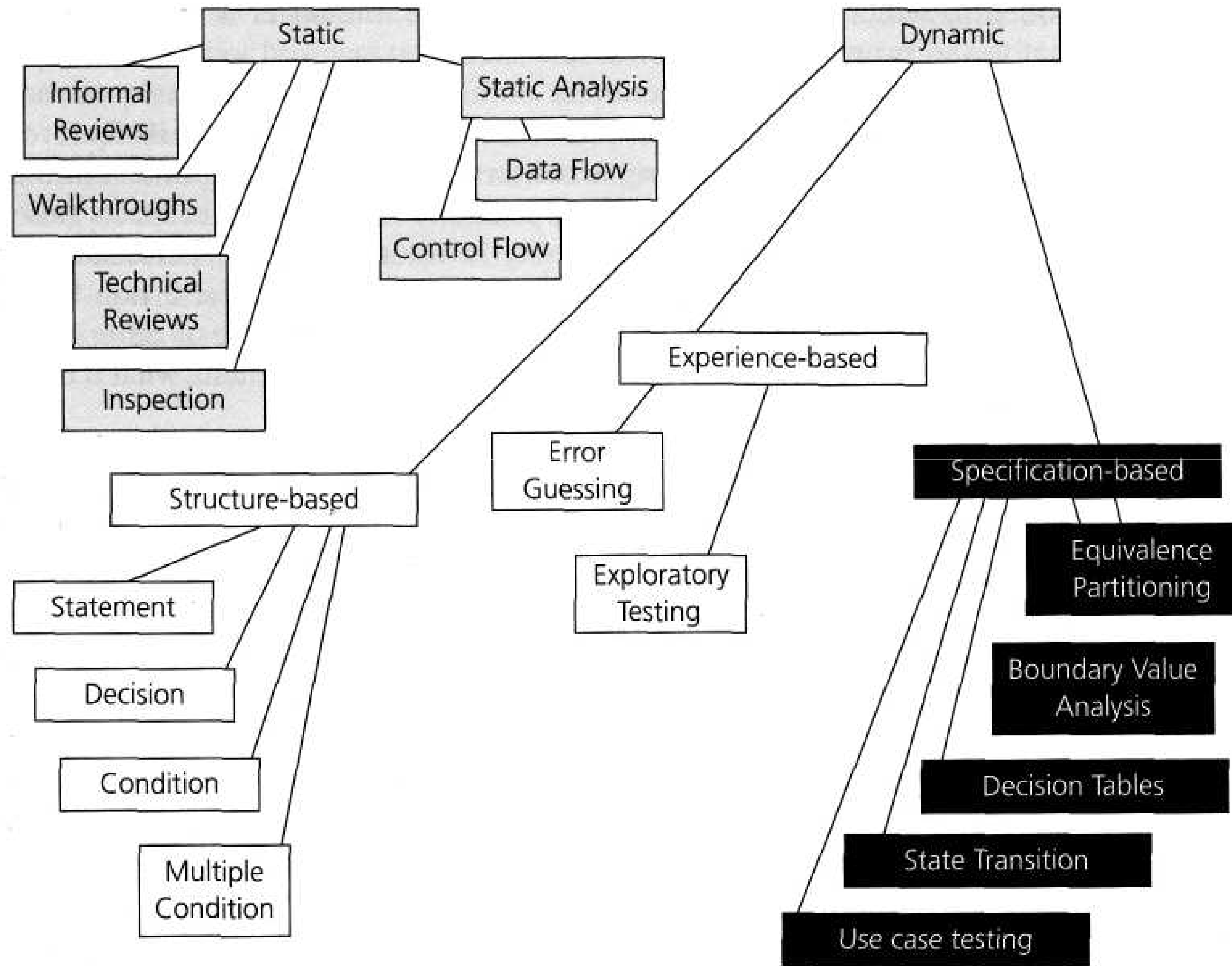


Dinamikus teszt technológiák

- Kód, vagy rendszer működése szükséges a teszteléshez
  - Korai fázisból a követelmények vehetők csak át



Teszt technológiák



Fehér, szürke, fekete doboz

- Fehér doboz tesztek
  - A kód belseje ismert a teszt folyamán
  - Tipikusan struktúra alapú tesztek
  - A V-modell alsó szakaszában elterjedt
- Fekete doboz tesztek
  - A belső működés részletei ismeretlenek csak a ki és bemenetekre fókuszál a teszt
  - Tipikusan specifikáció alapú tesztek
  - A V-modell egész tesztelési ágán alkalmazottak
- Szürke doboz tesztek
  - Alapvetően a ki és bemenetekre fókuszálunk a teljes belső struktúra nem ismert, de vannak információk, beavatkozási pontok a normál ki és bemeneteken kívül is.
  - Tipikus beágyazott rendszeres integrációs teszt technológia

Specifikáció alapú technikák

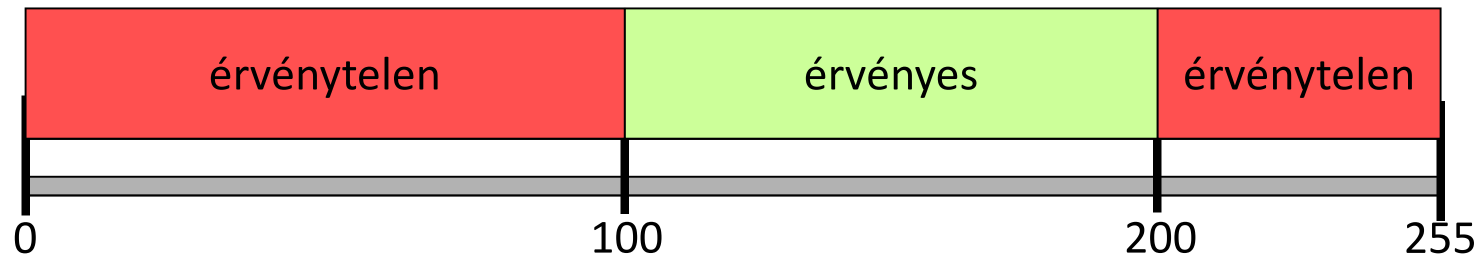
- Adott a tesztelendő funkcionális specifikációja, cél olyan reprezentatív teszt adatok keresése, amelyekkel a funkció a legkevesebb erőforrással tesztelhető (kimerítő tesztelés nem lehetséges)
- Módszerek
  - Ekvivalencia partíciók használata
  - Határérték ellenőrzés
  - Ok-hatás analízis, döntési táblák
  - Állapot átmenet tesztelés
  - Használati eset alapú tesztelés

Ekvivalencia partíciók használata

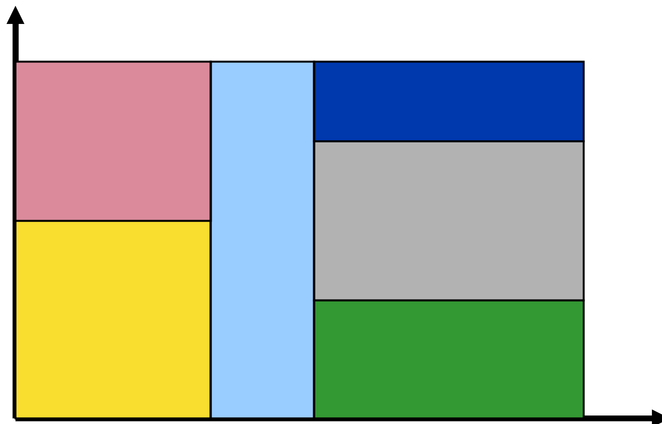
- Cél: Ekvivalencia osztályok azonosítása
  - Olyan bementi adatcsoportok keresése, amelyek vélhetően **azonos belső utat járnak be és azonos eredményt produkálnak**
  - Feltételezés: ha ezekből a csoportokból **egy teszt adatra helyesen/rosszul viselkedik a rendszer, akkor az összesre ugyanúgy fog**
  - Cél legalább egy adat minden partíciókból
- Tipikus partíciók
  - Érték tartományok
  - Halmazok
  - Egyedi kategóriák

Ekvivalencia partíciók használata

- Példák ekvivalencia partíciókra
  - Egyszerű értéktartományok

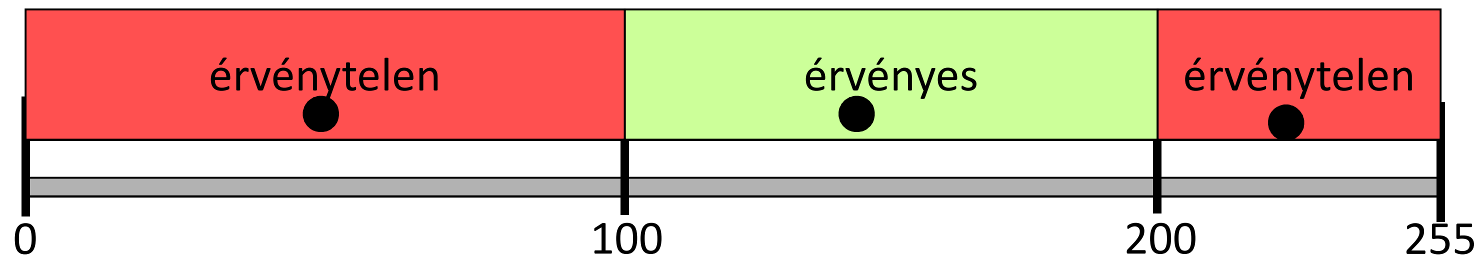


  - Bonyolultabb osztályok

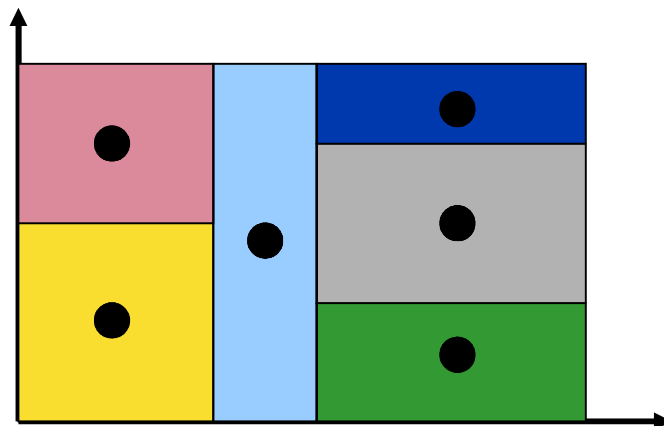


Ekvivalencia partíciók használata

- Példák ekvivalencia partíciókra
  - Egyszerű értéktartományok

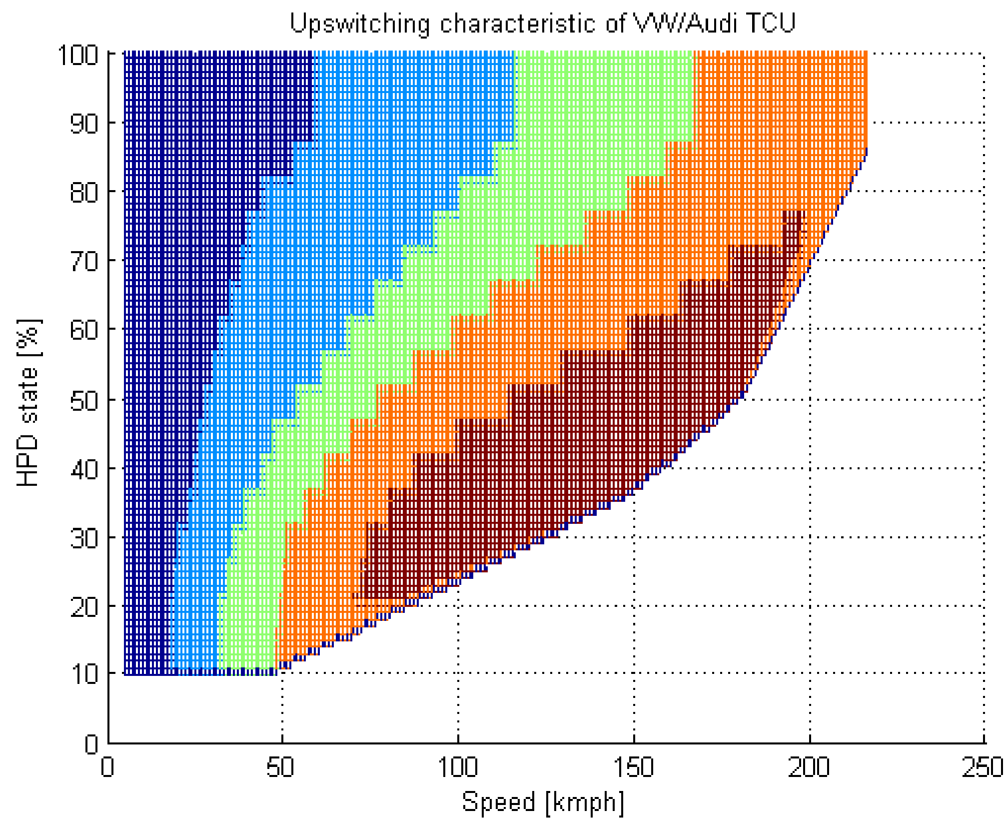


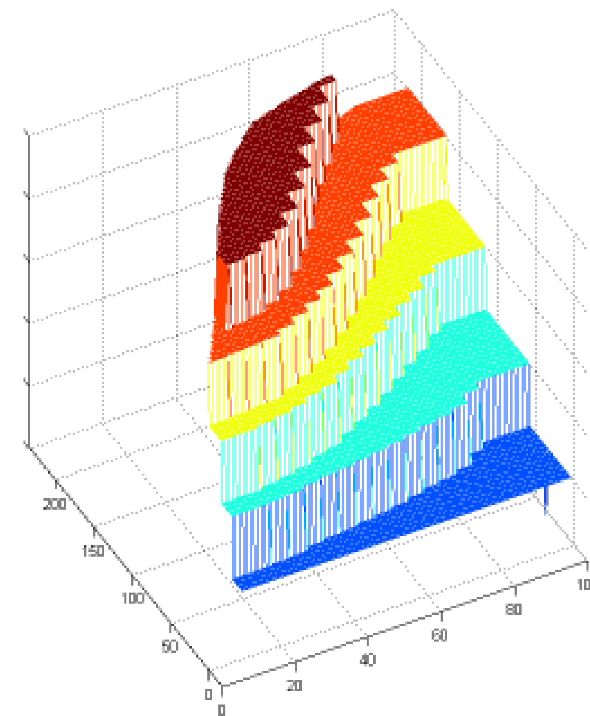
  - Bonyolultabb osztályok (itt is általában megjelennek az érvényes és nem érvényes osztályok)



Ekvivalencia partíciók példa

- Váltó karakterisztika

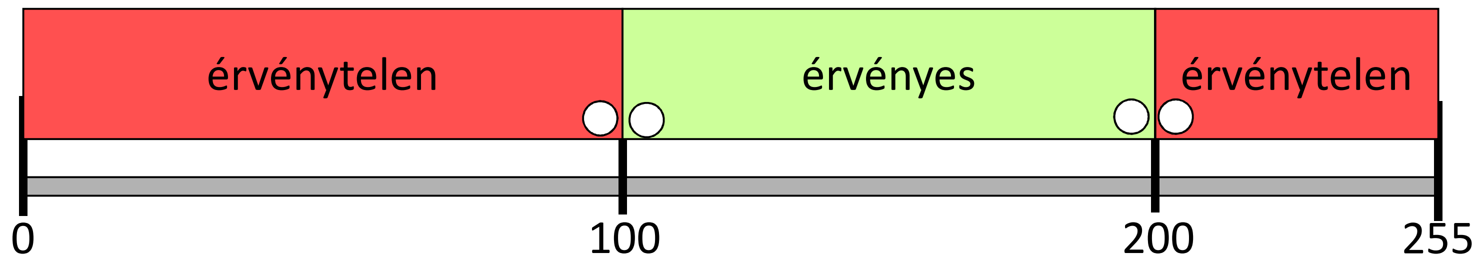




Határérték ellenőrzés

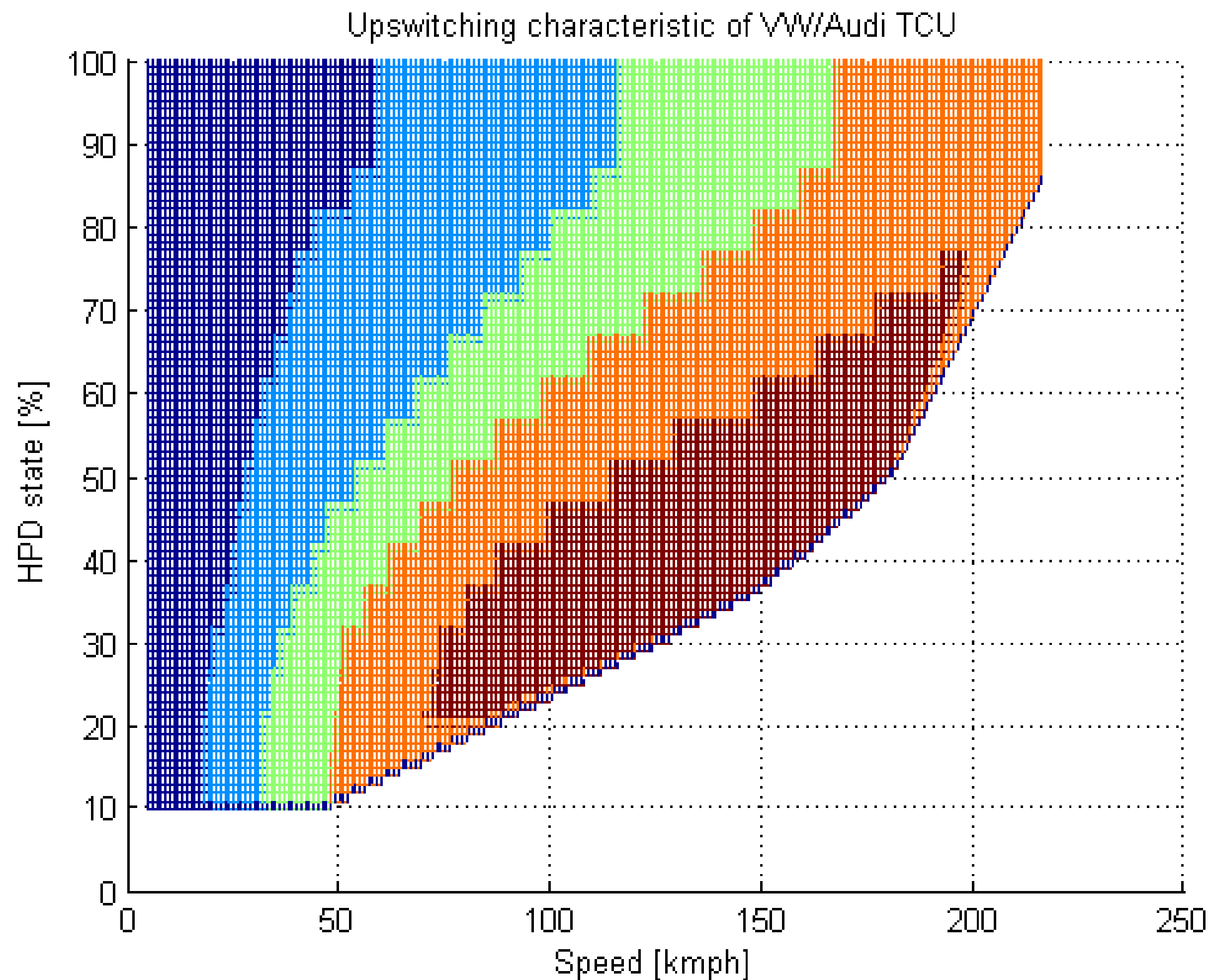
- A hibák legtöbb esetben a partíciók határánál rejtőnek
  - Hibás döntési relációk
  - Hibás ciklus be és kilépési feltételek
  - Hibás adatstruktúra kezelések

- Megoldás a határ mindkét oldalán teszteljünk



Ez azért nem feltétlenül olyan triviális

- Váltó karakterisztika



Miért csináljunk határérték és ekvivalencia partíció tesztet is?

- Ha csak határérték tesztet csináltunk, akkor lefedtük az egész partíciót?

Miért csináljunk határérték és ekvivalencia partíció tesztet is?

- Ha csak határérték tesztet csináltunk, akkor lefedtük az egész partíciót?
  - Technikailag igen és rendben is lehet ha minden ideálisan működik
  - Ha hibát jelent a teszt akkor az egész partíció viselkedése rossz, vagy csak a határérték van rossz helyen?
    - Mindenképpen tesztelni kell a partíció belsejét is
  - Csak az extrém szituációk tesztelése nem ad magabiztos hitelességet a normál szituációk megfelelő viselkedésre (főleg a felhasználók számára)
  - A határokat néha nagyon nehéz meghatározni

Ok-hatás analízis, döntési táblák

- Bementek és kimenetek kapcsolatának vizsgálata
  - **OK:** egy-egy bemeneti ekvivalencia osztály
  - **Hatás:** egy-egy kimeneti ekvivalencia osztály
  - Logikai változókat építünk ezekre
    - Sebesség > 50km/h pl.
    - Gázpedál állása < 30%
- Bool-gráf
  - ÉS, VAGY kapcsolatok
  - Meg nem engedett kombinációk
- Cél a logikai hálózat igazságtáblájának lefedése
  - Egy oszlop a táblázatban egy teszt esetnek felel meg

Ok-hatás analízis, döntési táblák: példa

- Diagnosztikai hozzáférés beágyazott vezérlőhöz

- Bemeneti adatok
  - Felhasználói név
  - Login
  - Kiterjesztett hozzáférési mód

- Kimeneti opciók
  - Diagnoszikai mód: Hibatároló kiolvasási lehetőség
  - Kiterjesztett diagnoszikai mód: Hibatároló törlési, szoftver frissítési mód

Ok-hatás analízis, döntési táblák: példa

- Diagnosztikai hozzáférés beágyazott vezérlőhöz
- Bemenetek logikai kombinációja

Érvényes felhasználói név	T	T	T	T	F	F	F	F
Érvényes login	T	T	F	F	T	T	F	F
Engedélyezet Kiterjesztett hozzáférési mód	T	F	T	F	T	F	T	F

- Bemeneti kombinációk racionalizálása

Érvényes felhasználói név	T	T	T	F
Érvényes login	T	T	F	-
Engedélyezet hozzáférési mód	T	F	-	-

Ok-hatás analízis, döntési táblák: példa

- Bemenetkhez tartozó válaszok (jogok)

Érvényes felhasználói név	T	T	T	F
Érvényes login	T	T	F	-
Engedélyezet hozzáférési mód	T	F	-	-
Hibatároló kiolvasás	T	T	F	F
Hibatároló törlés	T	F	F	F
Szoftver frissíés	T	F	F	F

Ok-hatás analízis, döntési táblák: példa

- Minden oszlophoz legalább egy teszt tartozik

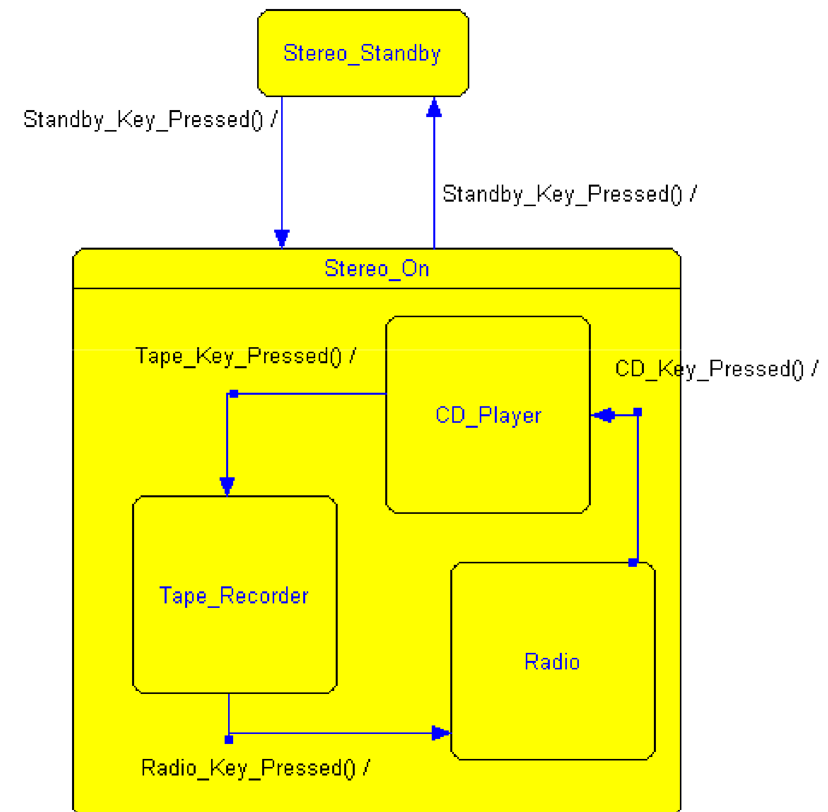
Érvényes felhasználói név	T	T	T	F
Érvényes login	T	T	F	-
Engedélyezet hozzáférési mód	T	F	-	-
Hibatároló kiolvasás	T	T	F	F
Hibatároló törlés	T	F	F	F
Szoftver frissíés	T	F	F	F
Minimum test case-ek	A	B	C	D

Ok-hatás analízis, döntési táblák

- Racionalizáció veszélyei
  - A racionalizálás mindig valamilyen feltételezésen alapul
  - A feltételezés lehet rossz
  - A feltételezéseket dokumentálni kell, mert az okok így a feltételek is változhatnak később
  - Az teljes igazságtábla használata segít a feltételezési hibák kiküszöbölésében, de nagyon megnöveli a teszt esetek számát, valahol a középút a jó megoldás

Állapot átmenet tesztelés

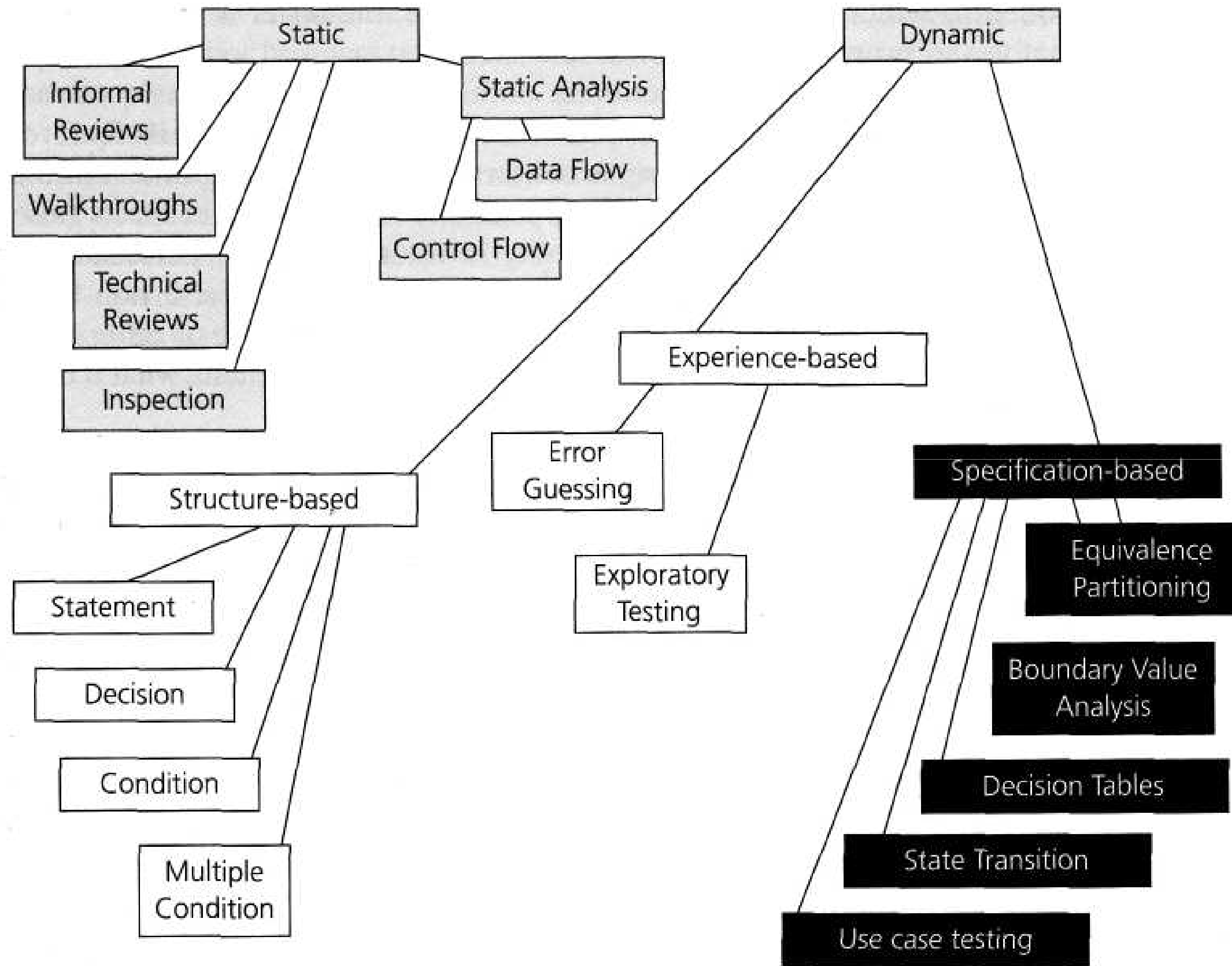
- A viselkedési modell egy véges automatával adott
- Tesztelési célok:
  - Minden állapot
  - Minden állapot átmenet vizsgálata
  - Nem megengedett átmenetek ellenőrzése



Használati eset alapú tesztelés

- Tipikusan a tesztelés késői szakaszában
- A felhasználási mintákból generált teszt eseteket végigjátszása a rendszeren

Teszt technológiák



© BME-MIT 2015

Struktúra alapú módszerek

- Minden esetben fehérdoboz módszerek: igénylik a szoftver végrehajtás valamilyen módszerrel történő megfigyelését
- Jól kezelhető mérőszámokat adnak
- Nem hibafedés
- Nem véd a nem megfelelő viselkedés ellen

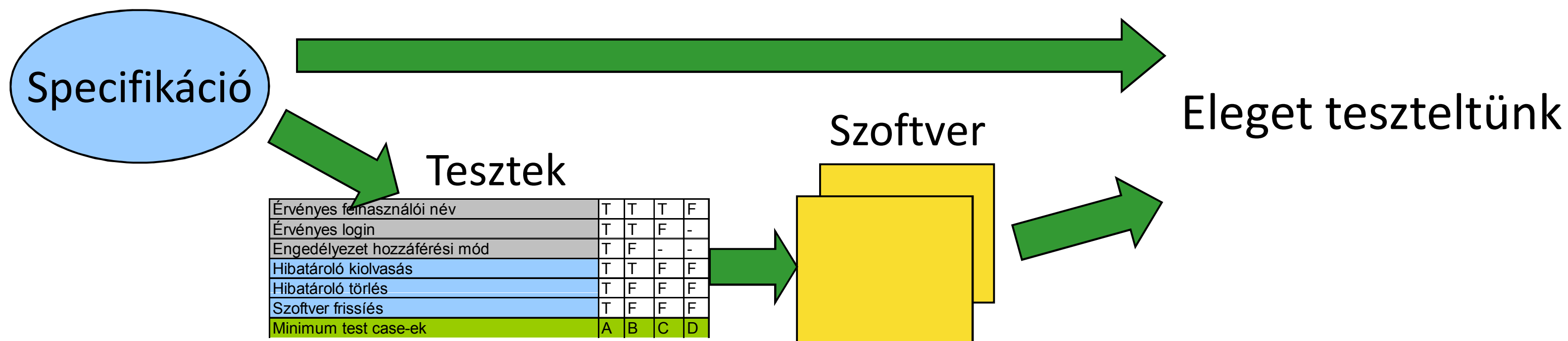
- Módszerek
  - Utasítás fedettség
  - Döntési ág fedettség
  - Feltétel fedettség
  - Útvonal fedettség

Alapfogalmak

- Utasítás: Satement
- Utasítás blokk: Block
  - Egybefüggő, elágazás, ugrás nélküli utasítások sorozata
- Feltétel: Condition
  - Egyszerű vizsgálat, amiben nincs logikai (Bool: AND, OR) operátor
- Döntés: Decision
  - Nulla vagy több logikai feltételből álló kifejezés
- Döntési ág: Branch
  - Egy döntés lehetséges kimenete
- Út: Path
  - Utasítások sorozata a modul be és kilépő pontja között

Miért használunk struktúra alapú módszereket?

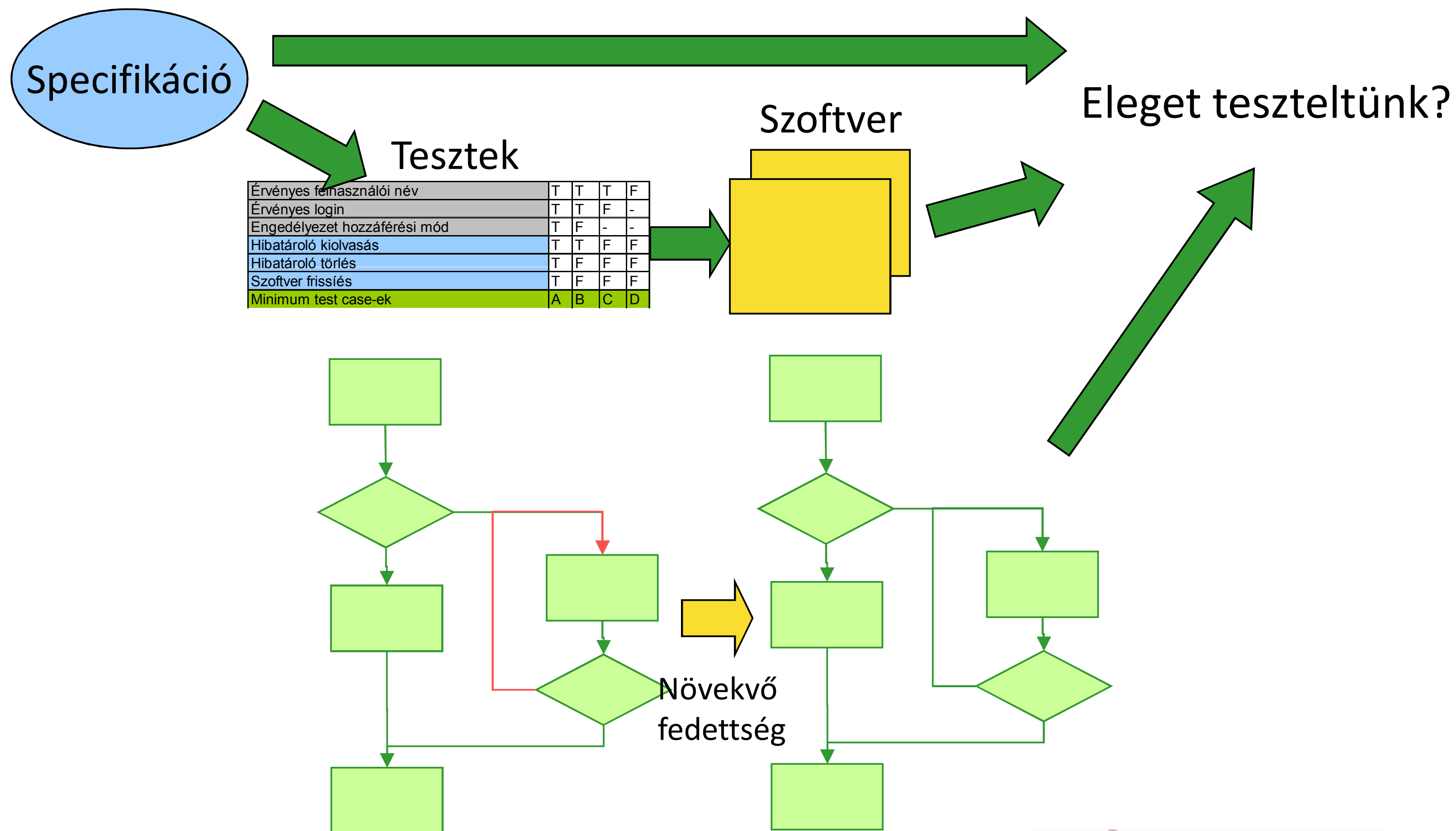
- Tipikus alkalmazás a szükséges tesztelés mennyiségének meghatározása



Érvényes felhasználói név	T	T	T	F
Érvényes login	T	T	F	-
Engedélyezet hozzáférési mód	T	F	-	-
Hibatároló kiolvasás	T	T	F	F
Hibatároló törlés	T	F	F	F
Szoftver frissíés	T	F	F	F
Minimum test case-ek	A	B	C	D

Miért használunk struktúra alapú módszereket?

- Tipikus alkalmazás a szükséges tesztelés mennyiségének meghatározása



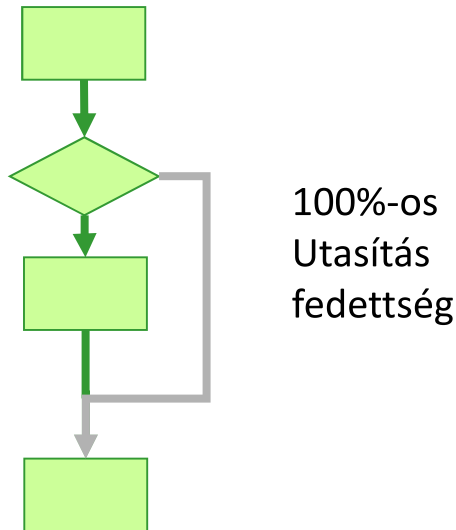
Érvényes felhasználói név	T	T	T	F
Érvényes login	T	T	F	-
Engedélyezet hozzáférési mód	T	F	-	-
Hibatároló kiolvasás	T	T	F	F
Hibatároló törlés	T	F	F	F
Szoftver frissíés	T	F	F	F
Minimum test case-ek	A	B	C	D

Utasítás fedettség (Statement coverage)

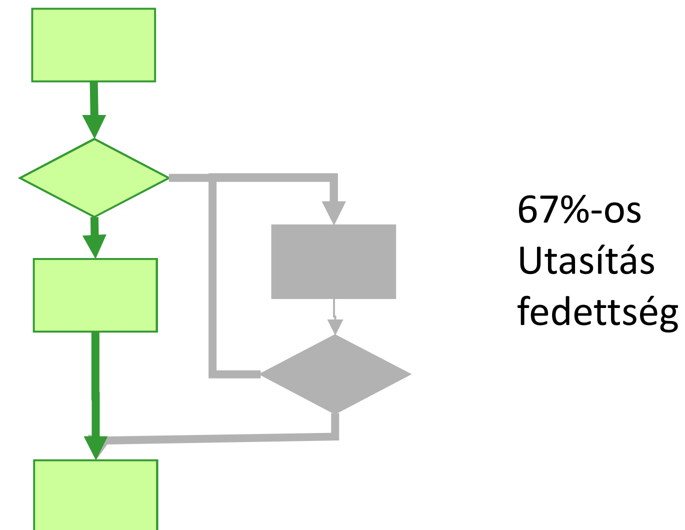
- Definíció:

$$\frac{\text{végrehajtott utasítások száma}}{\text{összes utasítás száma}}$$

Elég gyenge fedést ad



100%-os Utasítás fedettség



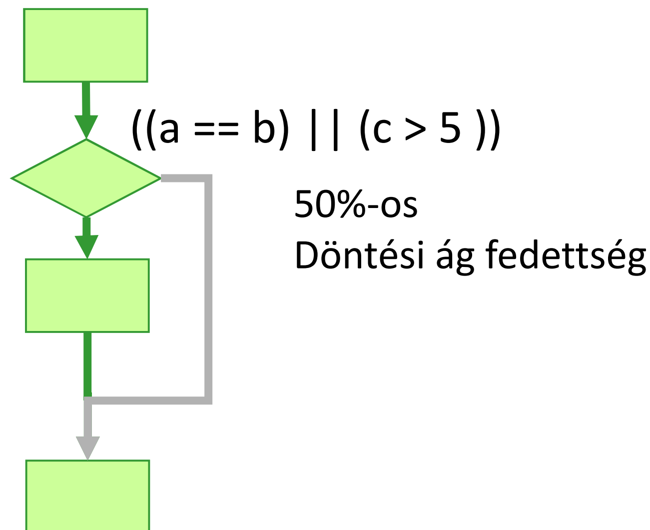
67%-os Utasítás fedettség

Döntési ág fedettség (Decision coverage)

- Definíció:

$$\frac{\text{a végrehajtott döntési ágak száma}}{\text{az összes döntési ág száma}}$$

- Viszonylag jó fedettség, de nem veszi figyelembe a feltétel kombinációkat

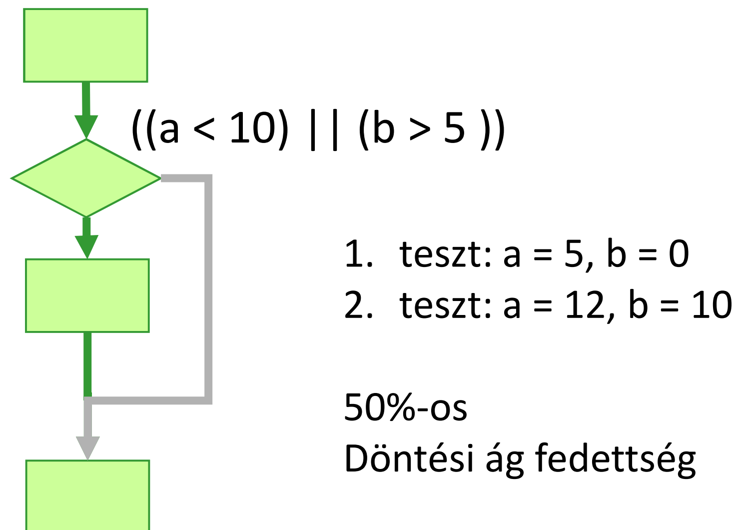


Feltétel fedettség (Condition coverage)

- Definíció:

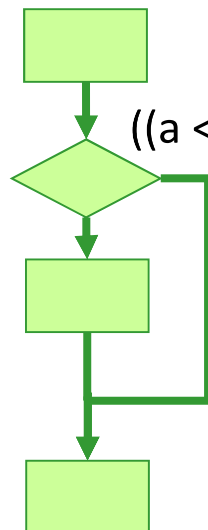
$$\frac{\text{feltételek tesztelt kombinációinak száma}}{\text{a feltételek megcélzott kombinációinak száma}}$$

- Mi az a megcélzott kombináció?
  - Az nem elég, ha minden feltétel felveszi az igaz és hamis értékét is egyszer, mert ez nem feltétlenül okoz 100%-os döntési fedettséget



Minden feltétel kombináció (Multiple Condition Coverage)

- Definíció:
  - A feltételek minden lehetséges kombinációja tesztelésre kerül
  - Feltételek számával exponenciálisan növekvő teszt eset



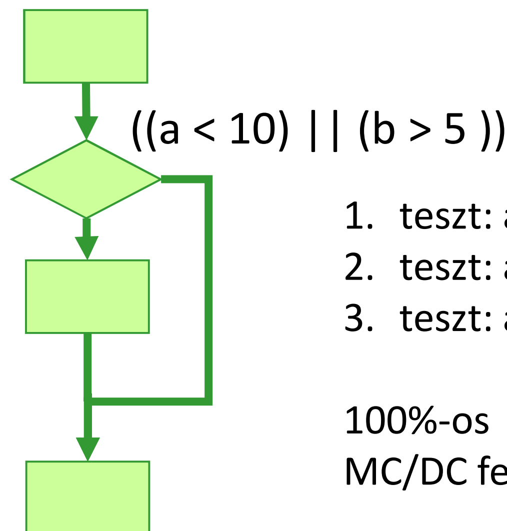
((a < 10) || (b > 5))

1. teszt: a = 5 (T), b = 0 (F)
2. teszt: a = 12 (F), b = 10 (T)
3. teszt: a = 15 (F), b = 2 (F)
4. teszt: a = 6 (T), b = 8 (T)

100%-os
Multiple Condition Coverage

Módosított feltétel/döntés fedettség (MC/DC coverage)

- Definíció:
  - Minden feltétel felveszi az összes lehetséges kimenetét egyszer
  - És minden döntés felveszi az összes lehetséges kimenetét egyszer
  - És minden feltétel a többitől függetlenül befolyásolja a hozzá tartozó döntés kimenetét
  - Nagyon gyakran alkalmazott fedettség definíció



1. teszt: a = 5 (T), b = 0 (F)
2. teszt: a = 12 (F), b = 10 (T)
3. teszt: a = 15 (F), b = 2 (F)

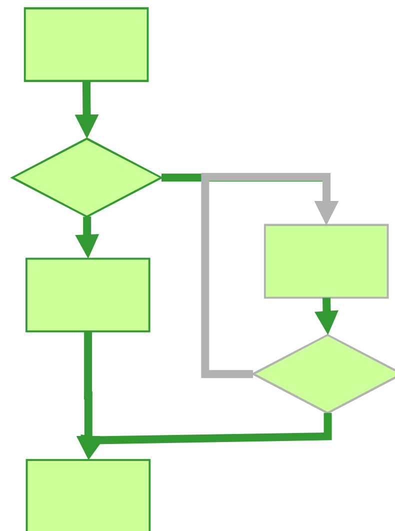
100%-os
MC/DC fedettség

Útfedettség (Path coverage)

- Definíció:

$$\frac{\text{végrehajtott független utak száma}}{\text{az összes független út száma}}$$

  - Független utak száma: Ciklomatikus komplexitás
  - Legerősebb fedettség mérték: nem szokták megkövetelni



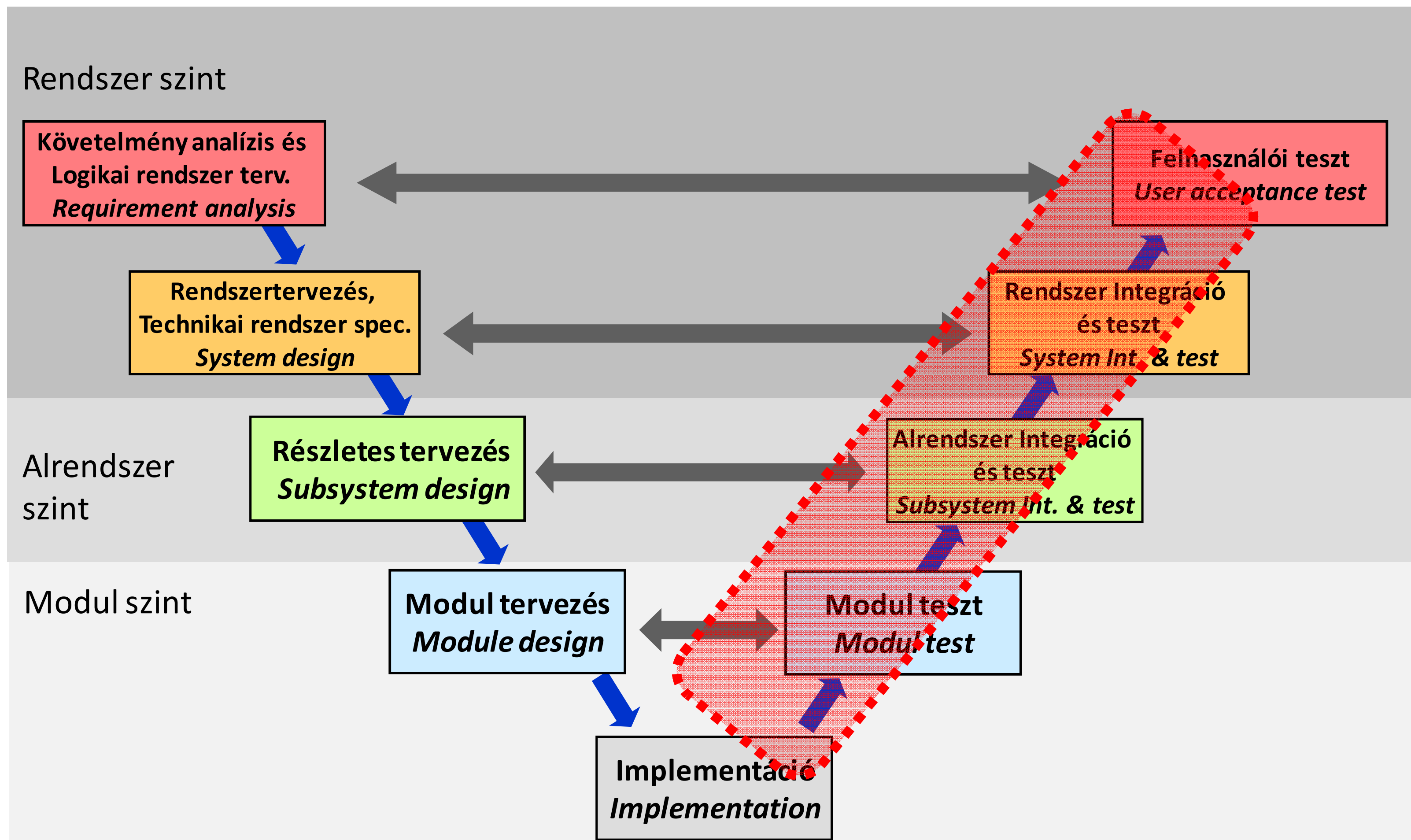
67%-os
Útfedettség

Nem szisztematikus tesztek

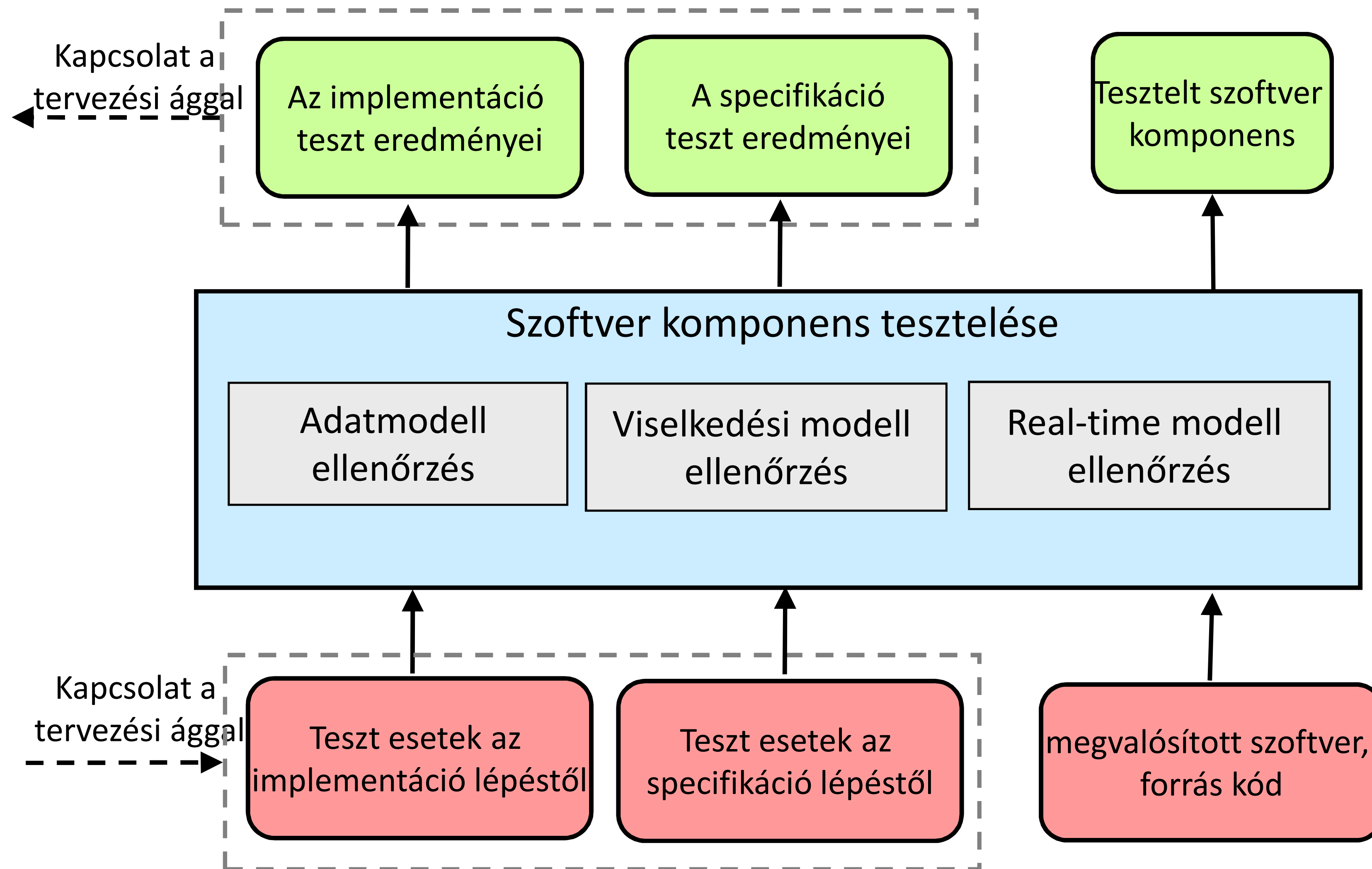
- Ad-hoc trial and error
  - Szisztéma nélküli próbálgatás
- Error guessing
  - A normál szisztematikus tesztek után szokták végrehajtani
  - Találhat olyan hibát, ami felett a szisztematikus tesztek elsiklanak
  - A teszt esetek a megérzésen, múltbeli hibákon, brain stormingon alapulnak
  - Az egyik mottó, hogy mi az a legőrültebb dolog, amit még csinálni tudunk
- User testing

Dinamikus tesztek a V-modellben

- Teszt technológiák alkalmazása



Szoftver komponens tesztek

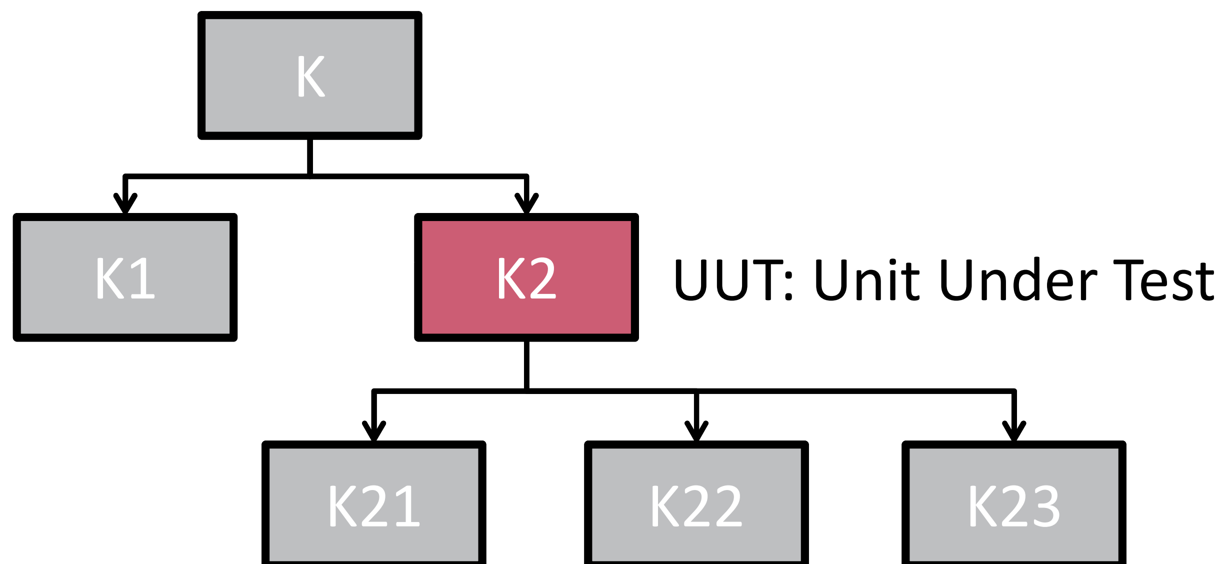


Modul tesztek jellemzői

- Cél a hibák gyors kijavítása a fejlesztés során
  - Tipikusan a fejlesztő végzi, így gyorsan kis költséggel kijavítható a hiba
- Modulok külön egységként kezelhetőek
  - Kis komplexitás
  - Könnyű a hiba lokalizációja
- Megkönnyíti az integrációt
  - A komplexitás kezelhetővé válik
- Sok esetben automatizálható
  - Sokszor kell végrehajtani, ezért gyorsnak kell lennie

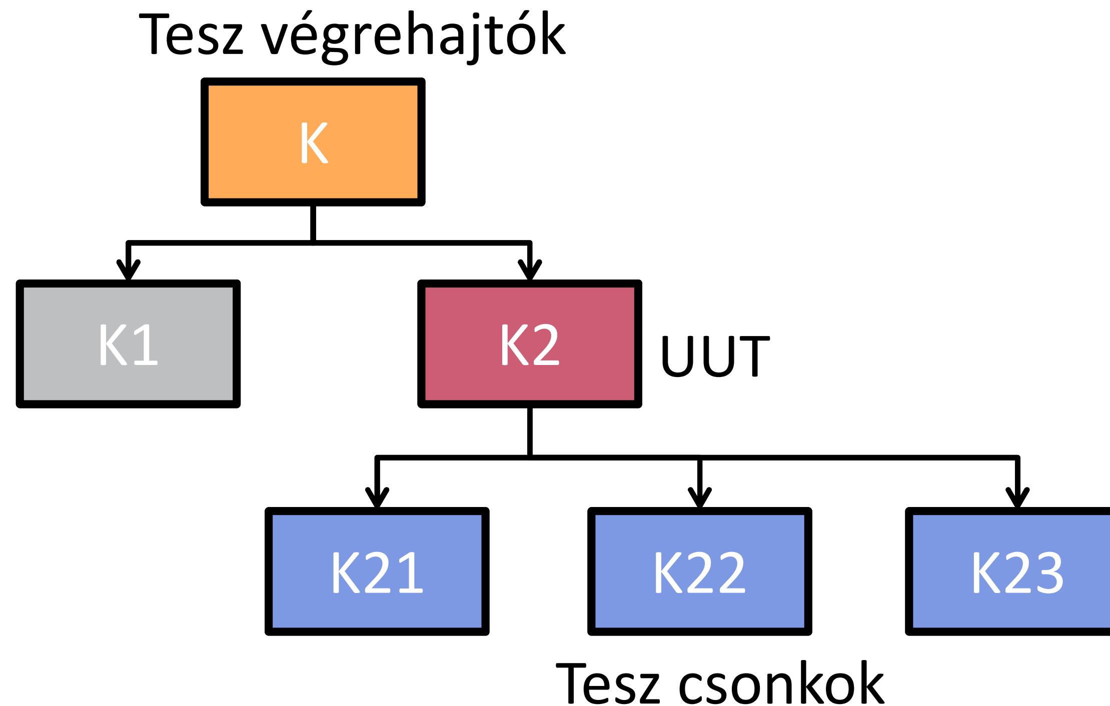
Modul tesztek tipikus problémája

- A modult önmagában kellene tesztelni, de az függ a környezetétől
  - Izoláció



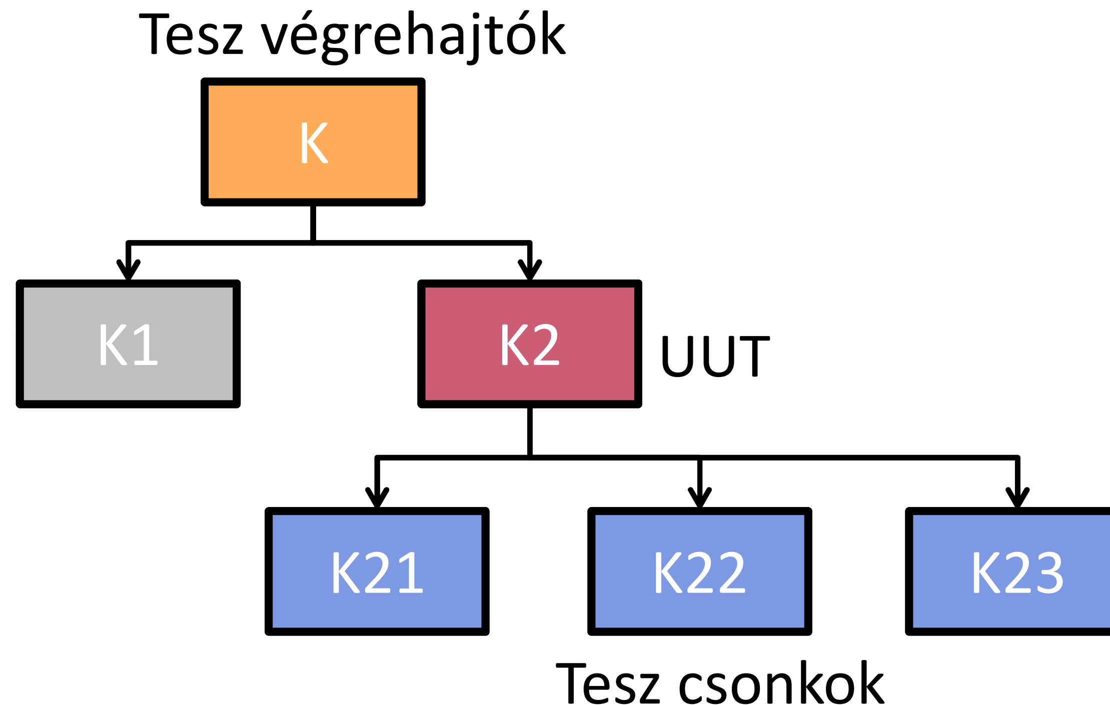
Modul tesztek tipikus problémája

- A modult önmagában kellene tesztelni, de az függ a környezetétől
  - Izoláció



Modul tesztek tipikus problémája

- A modult önmagában kellene tesztelni, de az függ a környezetétől
  - Izoláció



- Sokszor nem a tényleges hardware-en kerül sor a tesztelésre
  - Emuláció

Autóipar szabályozás a modul tesztre

	Methods	ASIL A	ASIL B	ASIL C	ASIL D
1a	Analysis of requirements	++	++	++	++
1b	Generation and analysis of equivalence classes	+	++	++	++
1c	Analysis of boundary values[a]	+	++	++	++
1d	Error guessing[b]	+	+	+	+

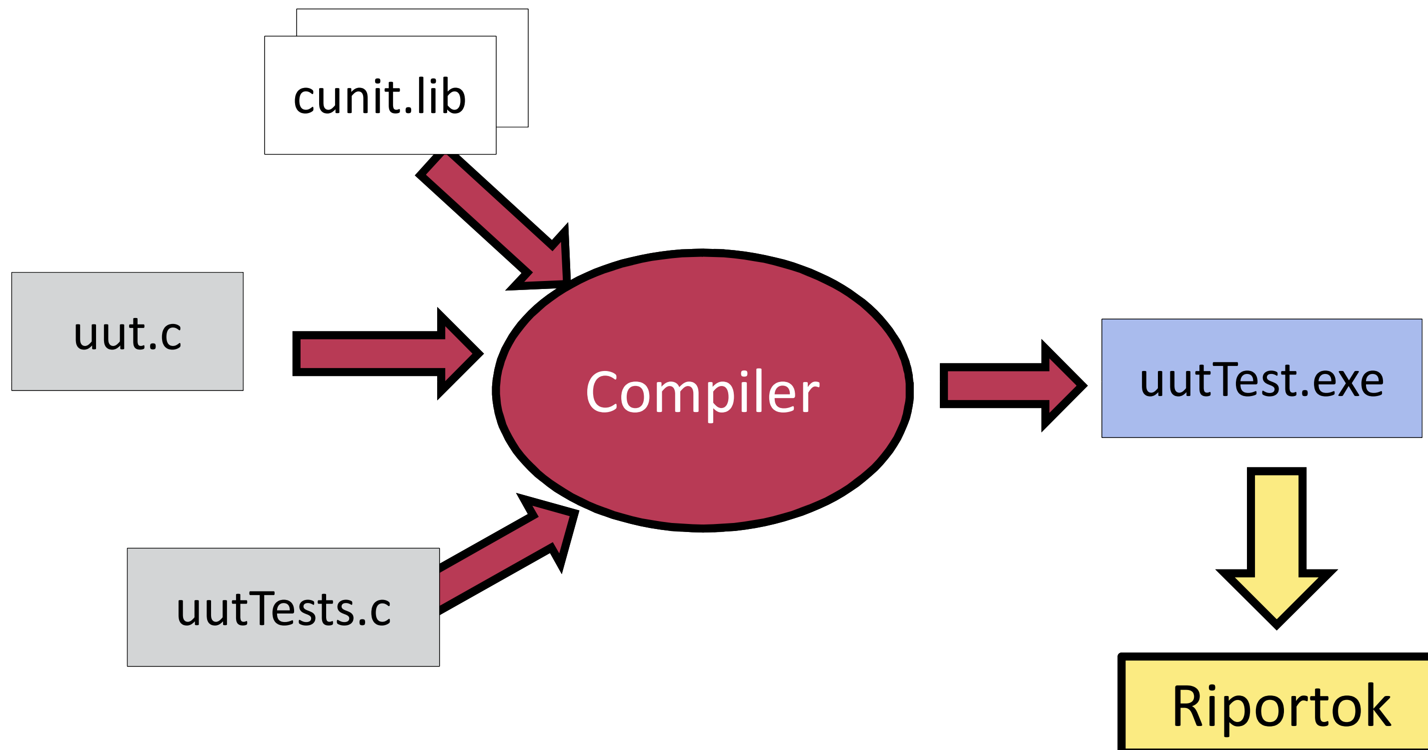
[a] This method applies to interfaces, values approaching and crossing the boundaries and out of range values.

[b] "Error guessing tests" can be based on data collected through a "lessons learned" process and expert judgment.

	Methods	ASIL A	ASIL B	ASIL C	ASIL D
1a	Statement coverage	++	++	+	+
1b	Branch coverage	+	++	++	++
1c	MC/DC (Modified Condition/Decision Coverage)	+	+	+	++

Specifikáció alapú vizsgálatok eszközei

- xUnit, cUnit: Egyszerű könyvtár automatikus unit teszteléshez



- Egyszerű assertion-ok használata: ***CU_ASSERT_TRUE(a), CU_ASSERT_EQUAL(a, b);***

Struktúra alapú vizsgálatok eszközei

- **Szoftveresen kód felműszerezéssel**
  - Kód felműszerezés overhead
    - Idő és program / adatmemória is
  - A release-ből sem szedhető ki a felműszerezés, mert módosíthatja a végrehajtás időt

Struktúra alapú vizsgálatok eszközei

- **Szoftveresen kód felműszerezéssel**
  - Kód felműszerezés overhead
    - Idő és program / adatmemória is
  - A release-ből sem szedhető ki a felműszerezés, mert módosíthatja a végrehajtás időt
- **Emulátorban való futtatással**
  - Nem szükséges felműszerezés
  - Nem a valós környezetben fut teljesen a kód
    - Hogy kezeljük a perifériákat, mi van az időzítésekkel
  - Debugger használata

Struktúra alapú vizsgálatok eszközei

- **Emulátorban való futtatással**
  - Nem szükséges felműszerezés
  - Nem a valós környezetben fut teljesen a kód
    - Hogy kezeljük a perifériákat, mi van az időzítésekkel
  - Debugger használata
- **Hardware-es méréssel**
  - Cím vagy adatvonal figyelés
    - Nehézkes modern hardware-nél legtöbbször nem külön chip
  - Mikrovezérlőbe integrált program és adatmemória esetén uC szintű támogatás szükséges hozzá
    - ARM Embedded trace module
  - Drága hardware interfész

Szoftveres tool pl.: GCOV

- GCC rész, vannak beágyazott próbálkozások is
- Felműszerezi a kódot és gyárt egy log file-t.
- A log file-ból tud következtetéseket levonni

```
#include <stdio.h>
int main (void)
         {
     1    int i;
    10    for (i = 1; i < 10; i++)
            {
     9        if (i % 3 == 0)
     3          printf ("%d is divisible by 3\n", i);
     9        if (i % 11 == 0)
######          printf ("%d is divisible by 11\n", i);
     9      }
     1    return 0;
     1  }
```

Hardware-es méréssel

- **Hardveres coverage tool**

[B::Data.ListHll /Cflag NoOK]

Step | Over | Next | Return | Up | Go | Break | Mode | Find: | arm.c

coverage	addr/line	source
ok	610	ast.field2 = 2;
ok	612	ast = func4(ast);
ok	614	j = (*funcptr)();
ok	616	start:
ok	617	j = func5((int) j
partial	619	if (j == 0)
never	620	goto start
ok	622	vfloat = 2.0;
ok	624	func6(vfloat, (fl
ok	626	vdouble = 2.0;
ok	628	func7(vdouble, (d
ok	630	func8();
ok	632	func9();

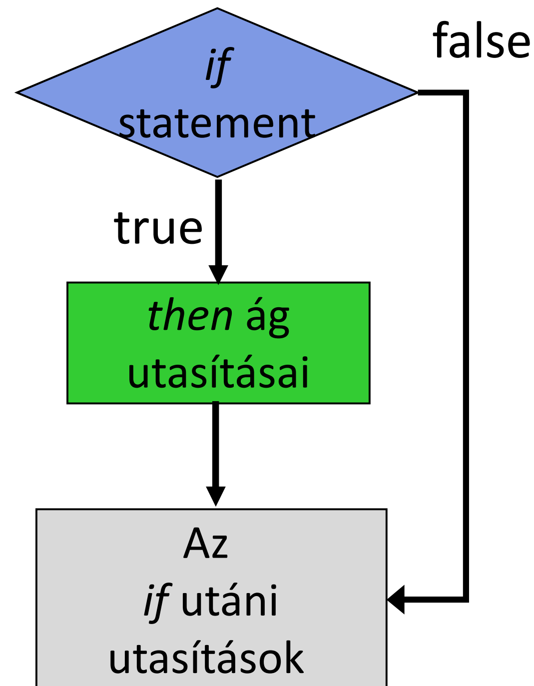
B::Trace.COVerage.ListFunc

Setup... | Goto... | List | Add | Load... | Save... | Init

address		coverage	executed	taken	nottaken
P:000010CC--00002377	a_li_aif	partial	88.033%	48.	46.
R:000010CC--000010D3	func0	never	0.000%	0.	0.
R:000010D4--000010E3	func1	ok	100.000%	0.	0.
R:000010E4--0000118F	func2	partial	93.023%	1.	1.
R:00001190--000011DF	func2a	ok	100.000%	1.	1.
R:000011E0--00001223	func2b	ok	100.000%	1.	1.
R:00001224--0000131B	func2c	partial	88.709%	1.	1.
R:0000131C--00001373	func2d	ok	100.000%	1.	1.
R:00001374--00001383	func3	partial	50.000%	0.	0.
R:00001384--000013BB	func4	partial	92.857%	0.	0.
R:000013BC--000013D3	func5	partial	66.666%	0.	0.
R:000013D4--00001447	func6	partial	89.655%	0.	0.
R:00001448--000014D3	func7	partial	74.285%	0.	0.
R:000014D4--0000176F	func8	partial	99.401%	0.	0.
R:00001770--00001803	func9	partial	89.189%	2.	2.
R:00001804--00001CFF	func10	partial	99.373%	33.	33.
R:00001D00--00001D7B	func11	partial	19.354%	1.	0.
R:00001D7C--00001DCB	func13	partial	90.000%	1.	1.
R:00001DCC--00001DE3	func14	partial	66.666%	0.	0.
R:00001DE4--00001DFF	func15	partial	71.428%	0.	0.
R:00001E00--00001E13	func16	partial	60.000%	0.	0.
R:00001E14--00001E37	func17	partial	77.777%	0.	0.
R:00001E38--00001E53	func18	partial	71.428%	0.	0.

Kérdések feltétel fedettségek esetében

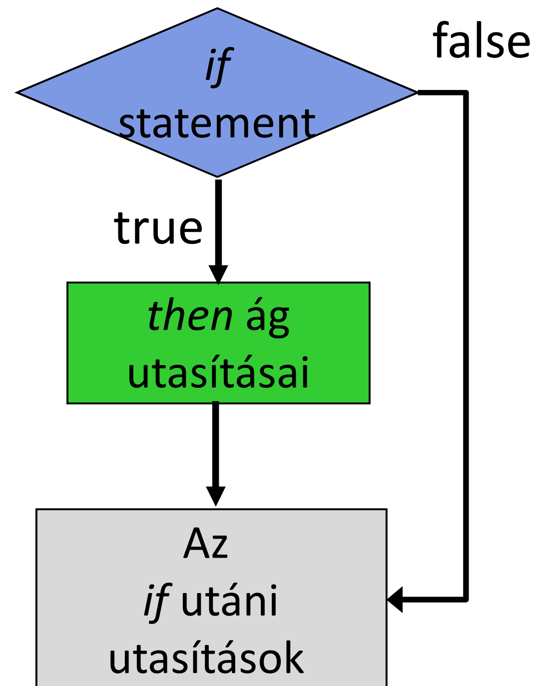
- Példa kód: `if ( (a == 10) && (b == 20 ))`



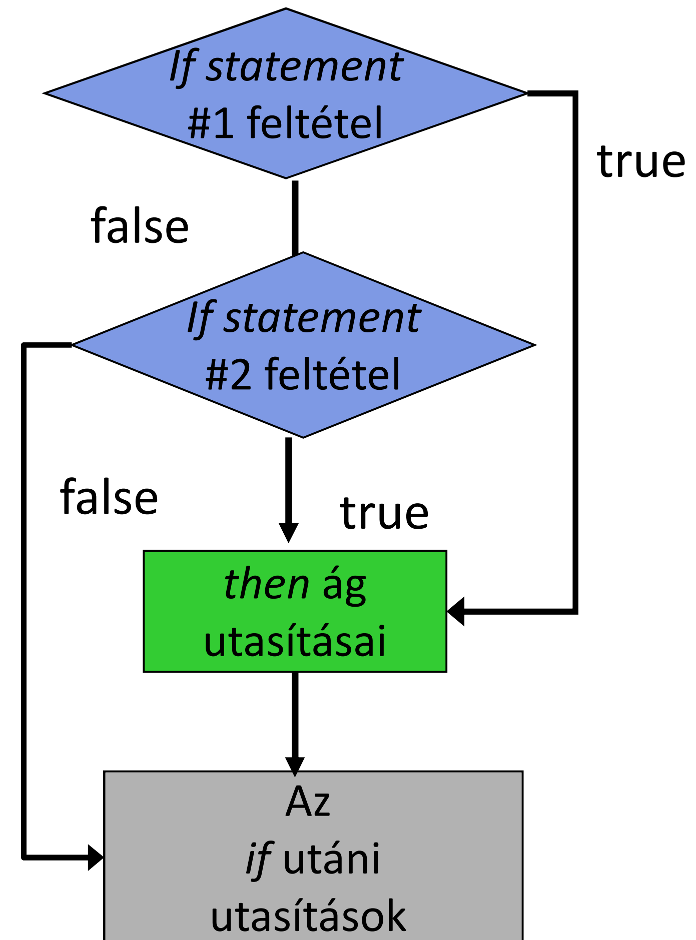
- Hogy mérünk itt feltétel fedettséget???

Kérdések feltétel fedettségek esetében

- Példa kód: if ((a == 10) && (b == 20))

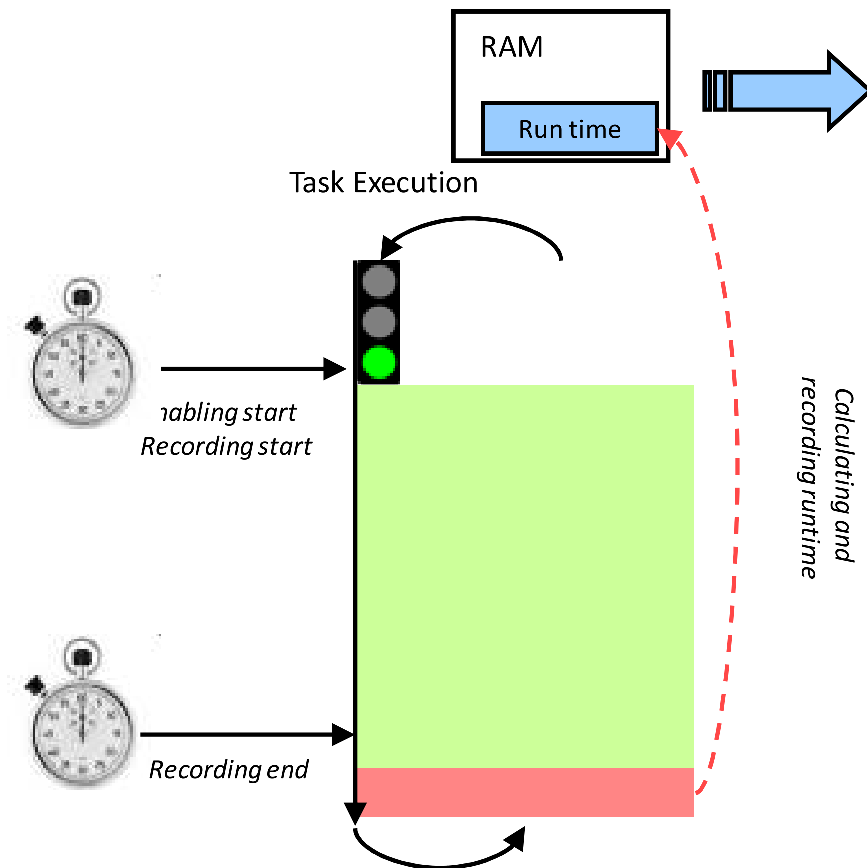


Basic Blokkokra lebontott kód



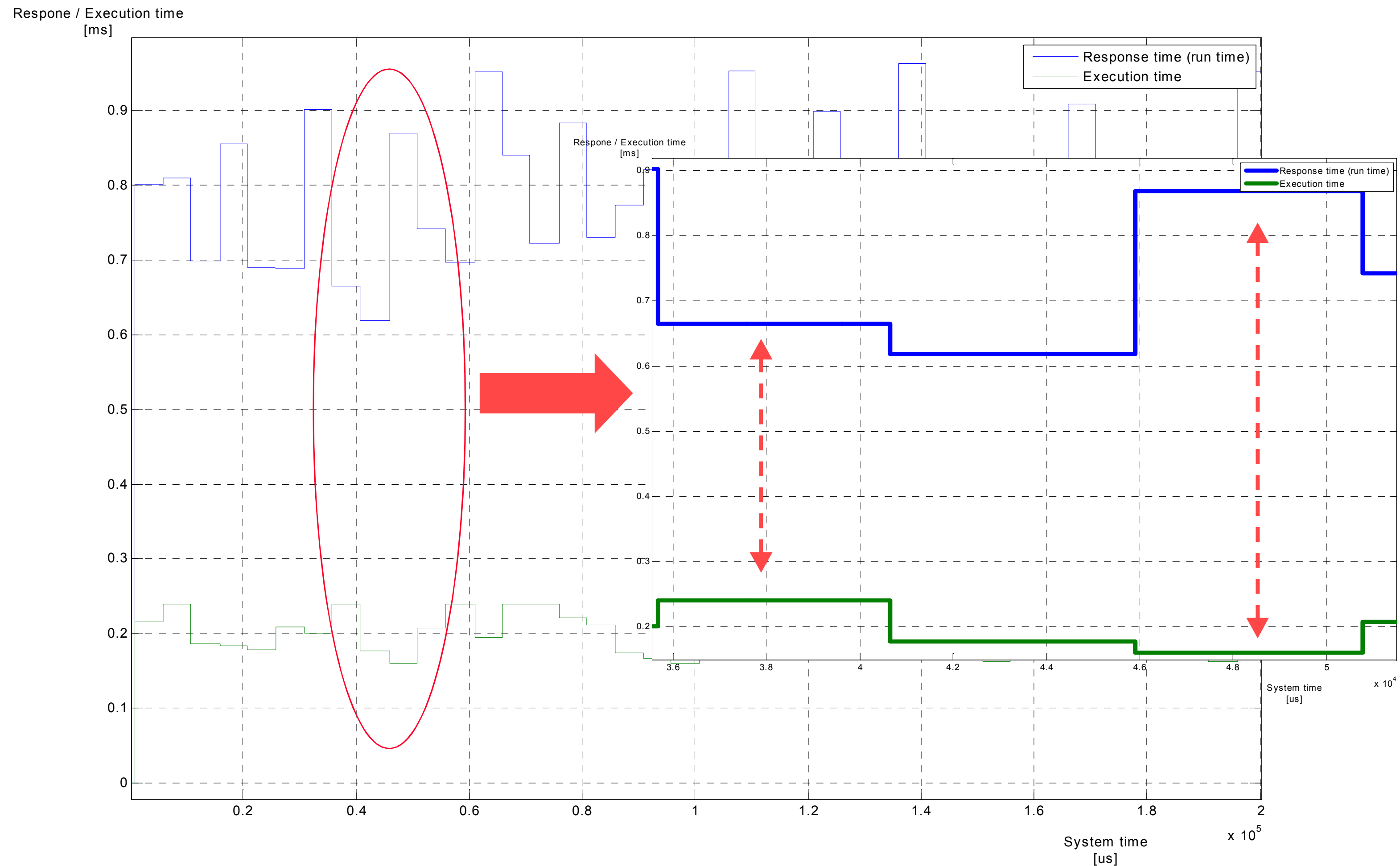
Real-time modell ellenőrzés

- Ütemezhetőségi vizsgálat
- Milyen információkra van szükségünk?
- Elég-e a szál indulás és vég mérése: **Válaszidő**



A válaszidő nagyon becsapós lehet

- A végrehajtási idő az igazán lényeges



A végrehajtási időből számítható minden

- **DMA:** Deadline Monotonic analysis

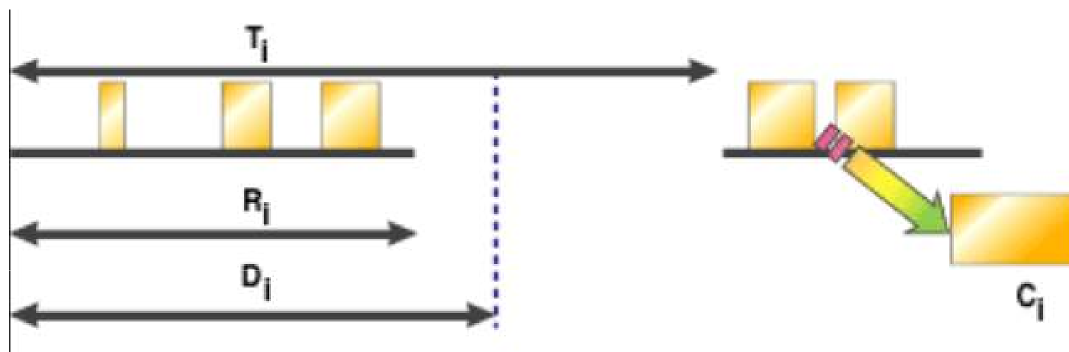


Table 1 DMA example

Task	T	C	D
1	250ms	5ms	10ms
2	10ms	2ms	10ms
3	330ms	25ms	50ms
4	1000ms	29ms	1000ms

$$R_i = C_i + \sum_{\forall k \in hp_i} \left\lceil \frac{R_i}{T_k} \right\rceil C_k$$

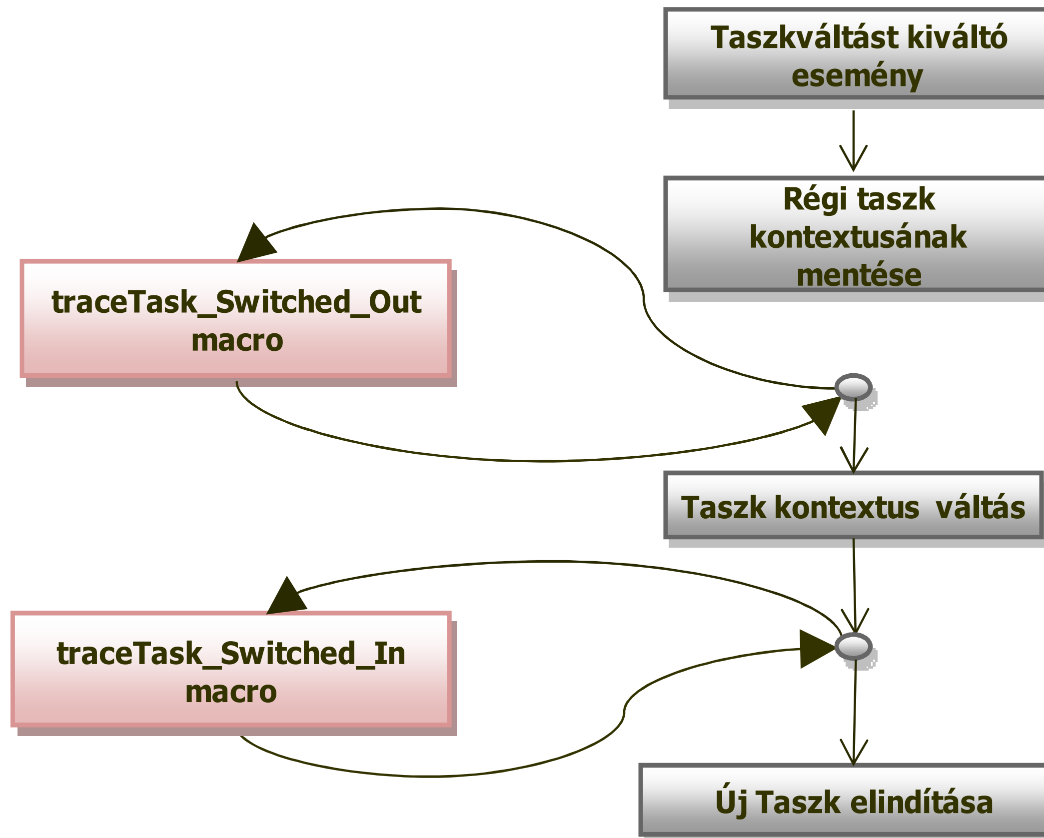
Table 2 Calculation of worst-case Task 3 response time

Step	R^n	I	R^{n+1}
1	0	0	25
2	25	5+3x2=11	36
3	36	5+4x2=13	38
4	38	5+4x2=13	38

- Hogyan mérjünk végrehajtási időt?

RTOS Trace hookok

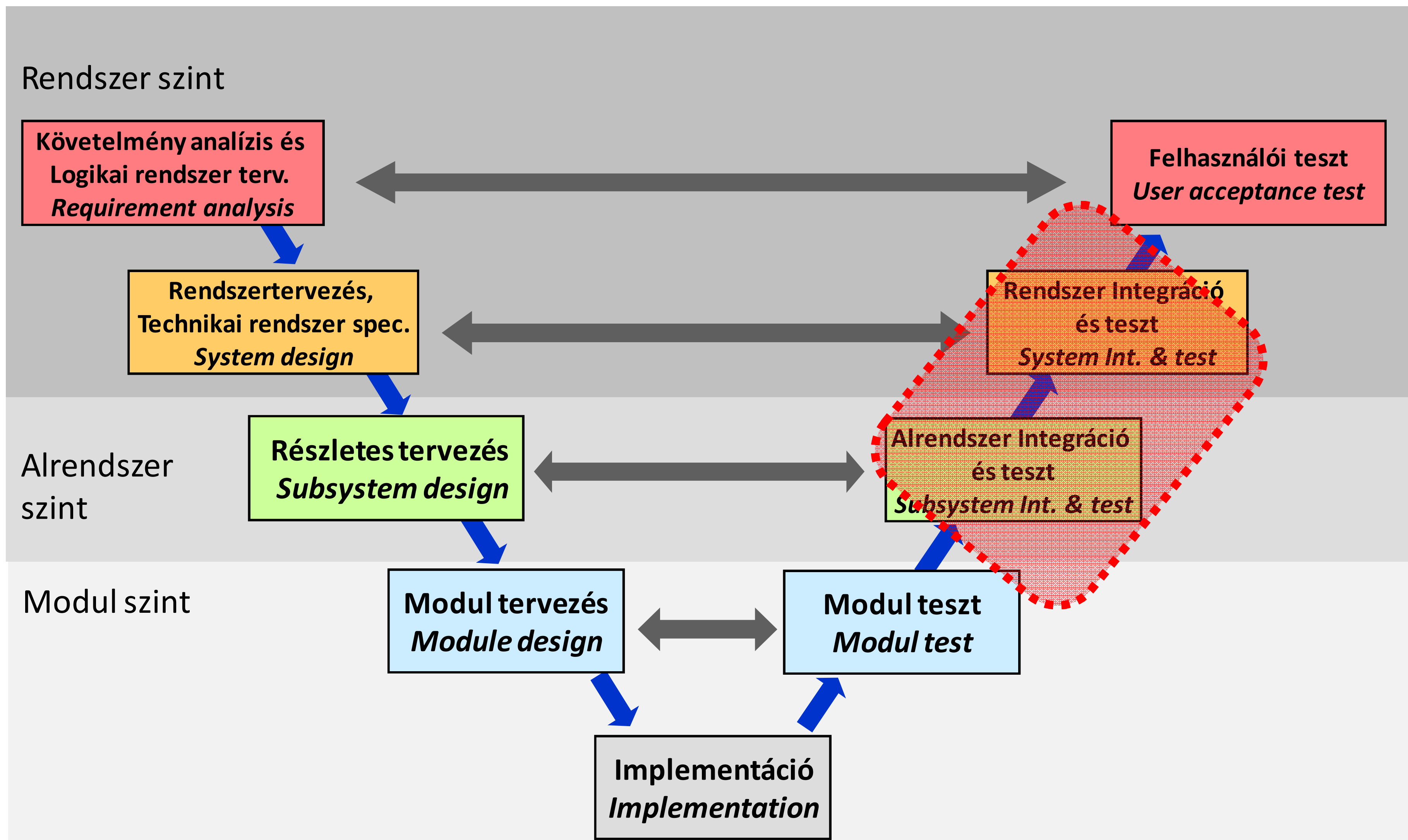
- FreeRTOS-nél gyakorlatilag minden lépéshez



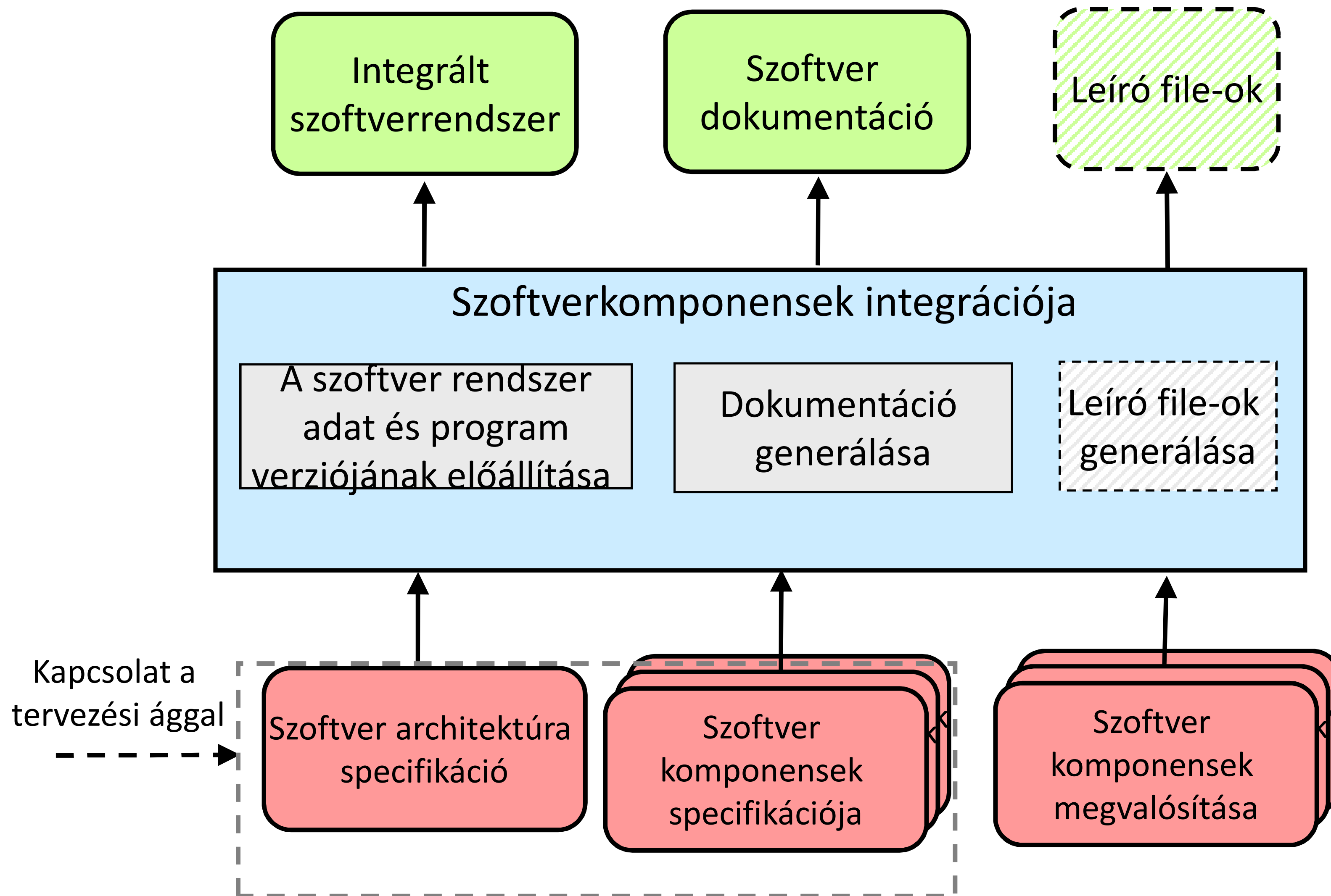
DWT: Data Watchpoint Trace

- 4 komparátor: data address / program counter (az első konfigurálható clock cycles counterre is) Van hozzá MASK is
  - Hardware watchpoint: processzor debug módba
  - ETM trigger: trace csomag küldés indítás
  - PC mintavételező trigger
  - Data address sample trigger
- Számlálók
  - Órajel számláló
  - Sleep ciklusokat számláló
  - Interrupt overhead számláló
- PC mintavételezés időközönként
- Interrupt trace

Integráció és integráció tesztelés



Szoftverrendszer integráció



Az integráció lehetséges módja: *Big-bang*

- Big-bang módszer
  - Elméletileg tesztelt komponensekkel rendelkezünk. Miért nem integráljuk és próbáljuk ki őket egyszerre, ezzel elméletileg időt nyerünk
  - Gyakorlatilag az esetleges hibák lokalizációja nehezebb
  - A hibák utáni újratesztelés nagyobb erőforrást igénylő folyamat
  - Tesztelés a külső interfészen keresztül valósul meg
  - Kis rendszer esetében azért praktikus lehet

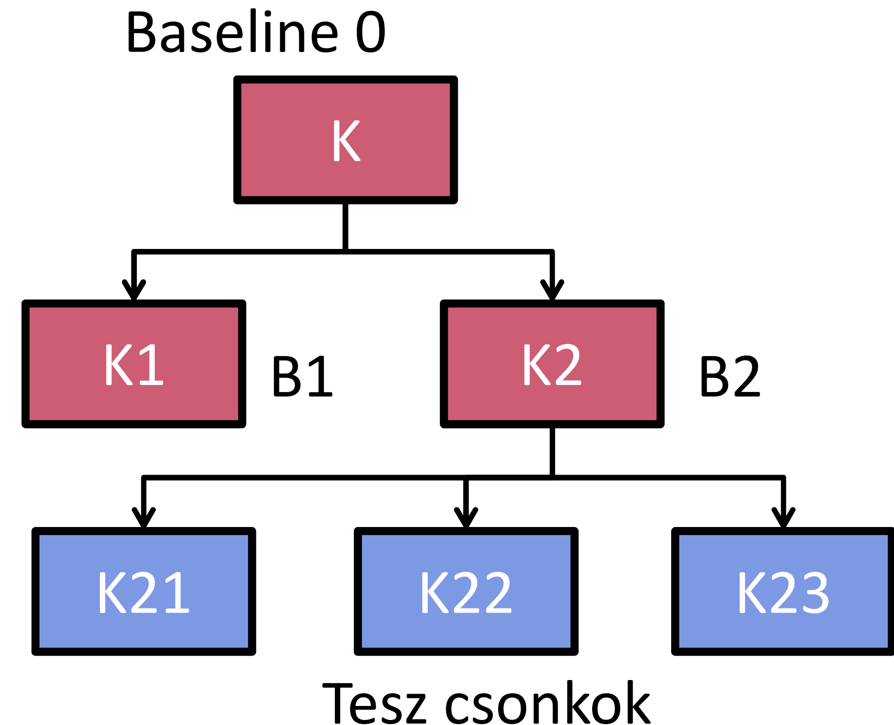
Inkrementális integráció

- Baseline 0: 1 tesztelt komponens
- Baseline 1: 2 tesztelt komponens
- Baseline 2: 3 tesztelt komponens …
- Előnyök
  - Könnyebb hiba lokalizáció
  - Könnyebb az esetleges problémákból helyreállítani a rendszert
  - Az interfészek elméletileg a komponens teszteknél letesztelődtek, de …

© BME-MIT 2015

Top-down integráció

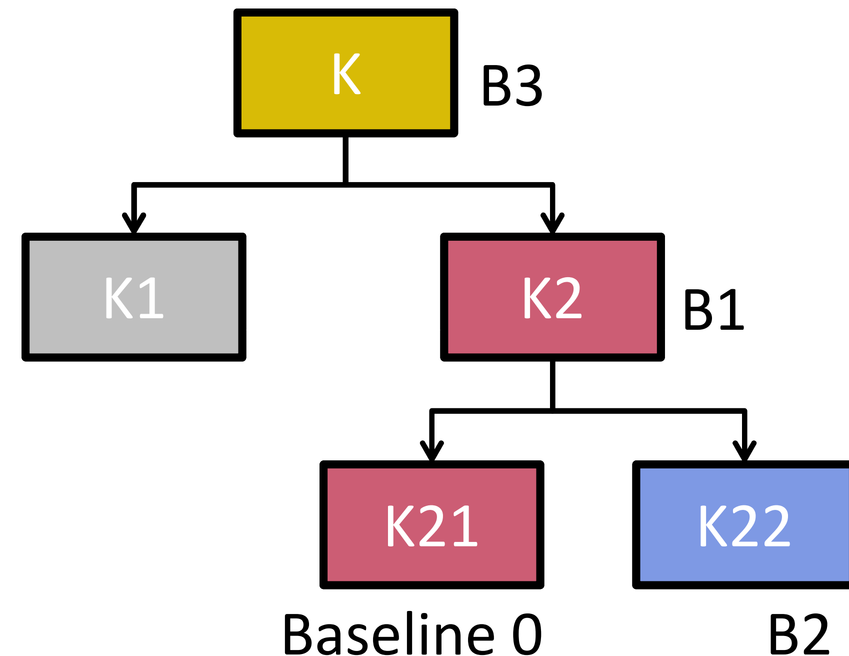
- Fentről lefelé haladunk
- Szükség van teszt csonkokra
  - Egyszerű printf
  - Várakozás
  - Konstans, vagy számított válasz
  - Táblázatból vett válasz
- Előnyök
  - A legmagasabb szintű funkciók futnak először (legkritikusabb, legjobban demonstrálható)
- Hátrányok
  - Szükség van teszt csonkokra
  - A részletek a végére maradnak
  - A végeredmény sokáig „szimulált”



© BME-MIT 2015

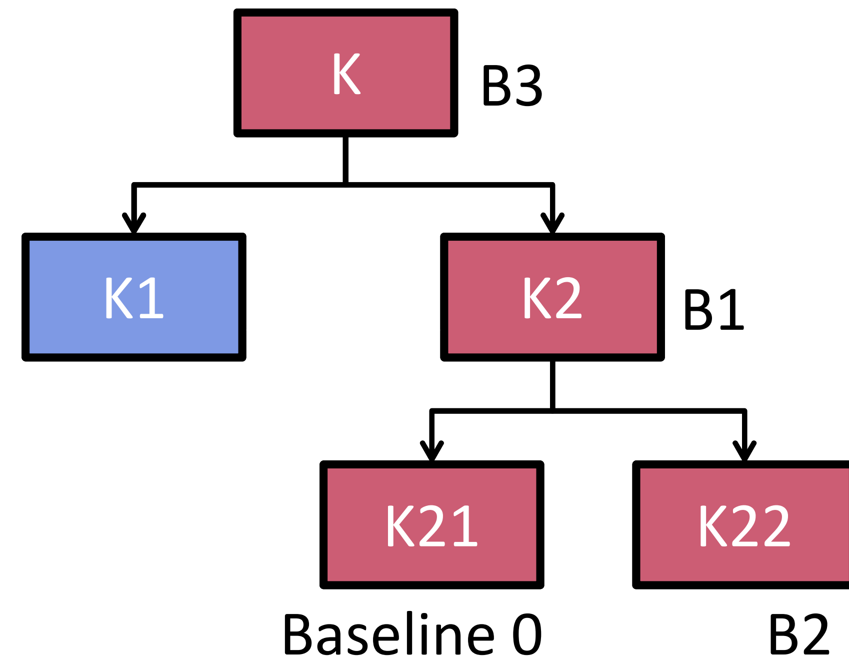
Botton-up integráció

- Fentről lefelé haladunk
- Szükség van teszt driverekre, és bizonyos helyeken teszt csonkra is
- Előnyök
  - Jó a hardware-hez kötődő rendszereknél
  - Szinte végig valós adatokkal, időzítésekkel dolgozunk
- Hátrányok
  - Csak az utolsó baseline-nál áll elő a működő rendszer
  - A felső szintű vezérlési gondok csak a végén derülnek ki
  - Teszt csonk és driver is kell



Botton-up integráció

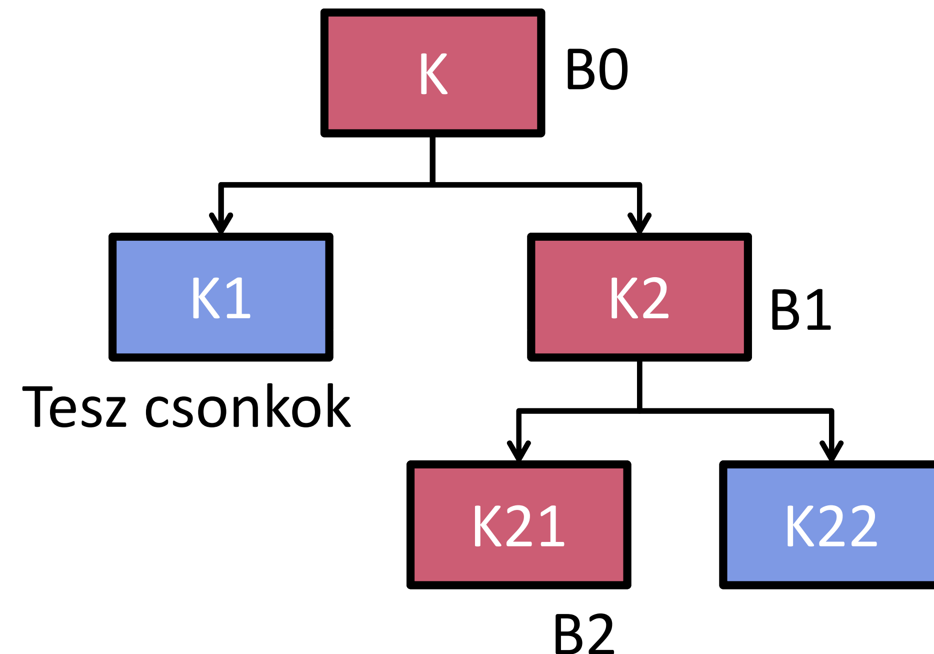
- Fentről lefelé haladunk
- Szükség van teszt driverekre, és bizonyos helyeken teszt csonkra is
- Előnyök
  - Jó a hardware-hez kötődő rendszereknél
  - Szinte végig valós adatokkal, időzítésekkel dolgozunk
- Hátrányok
  - Csak az utolsó baseline-nál áll elő a működő rendszer
  - A felső szintű vezérlési gondok csak a végén derülnek ki
  - Teszt csonk és driver is kell



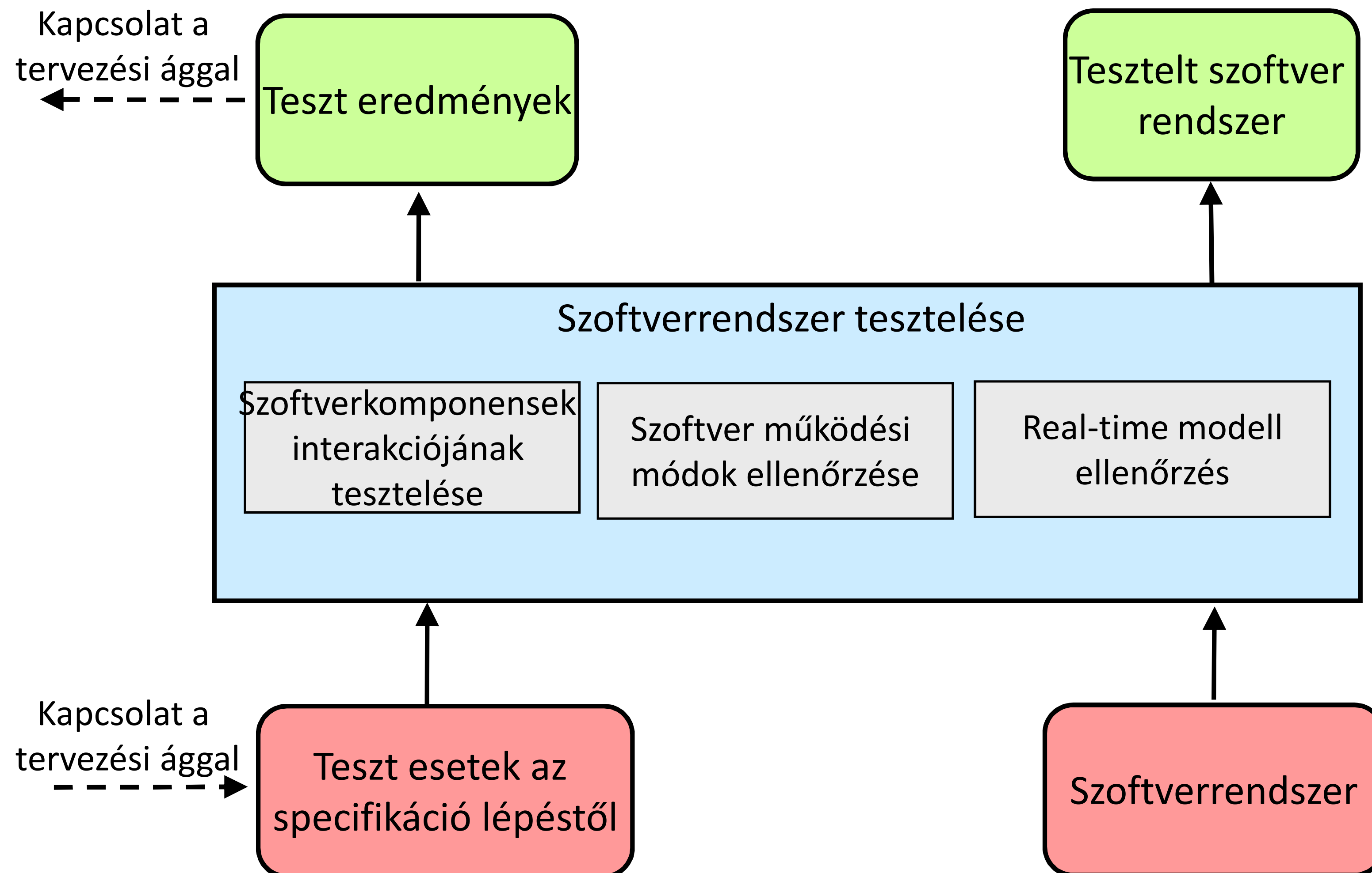
Tesz csonkok

Funkcionális integráció

- Egy funkció mentén haladunk fentről lefelé
- Szükség van sok teszt csonkra
- Előnyök
  - A legfelső szint van a legtöbbször és legrészletesebben tesztelve
  - Hamar dolgozhatunk valós eredményekkel is
  - Tényleges valós működő részrendszer nagyon hamar
- Hátrányok
  - Sok teszt csonk kell



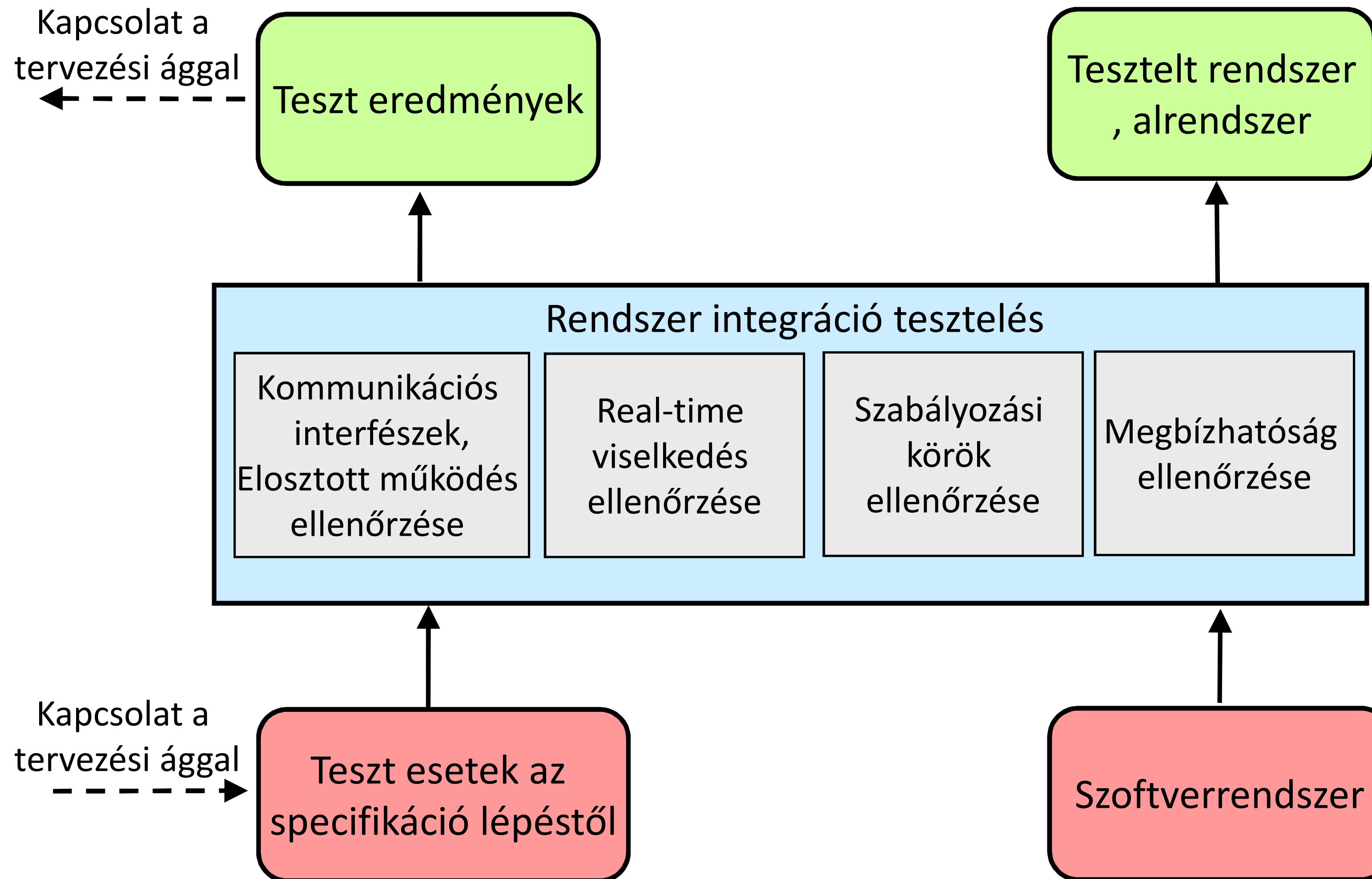
Szoftverrendszer tesztelése



Tipikusan alkalmazott integrációs tesztek

- Specifikáció alapú technikák
  - Ekvivalencia partíciók használata
  - Határérték ellenőrzés
  - Ok-hatás analízis, döntési táblák
  - Állapot átmenet tesztelés
  - Használati eset alapú tesztelés
- Nem funkcionális tesztek
  - Performance, load tesztek
  - Megbízhatóság tesztelés
  - Használhatóság tesztek

Rendszert integráció és tesztelés

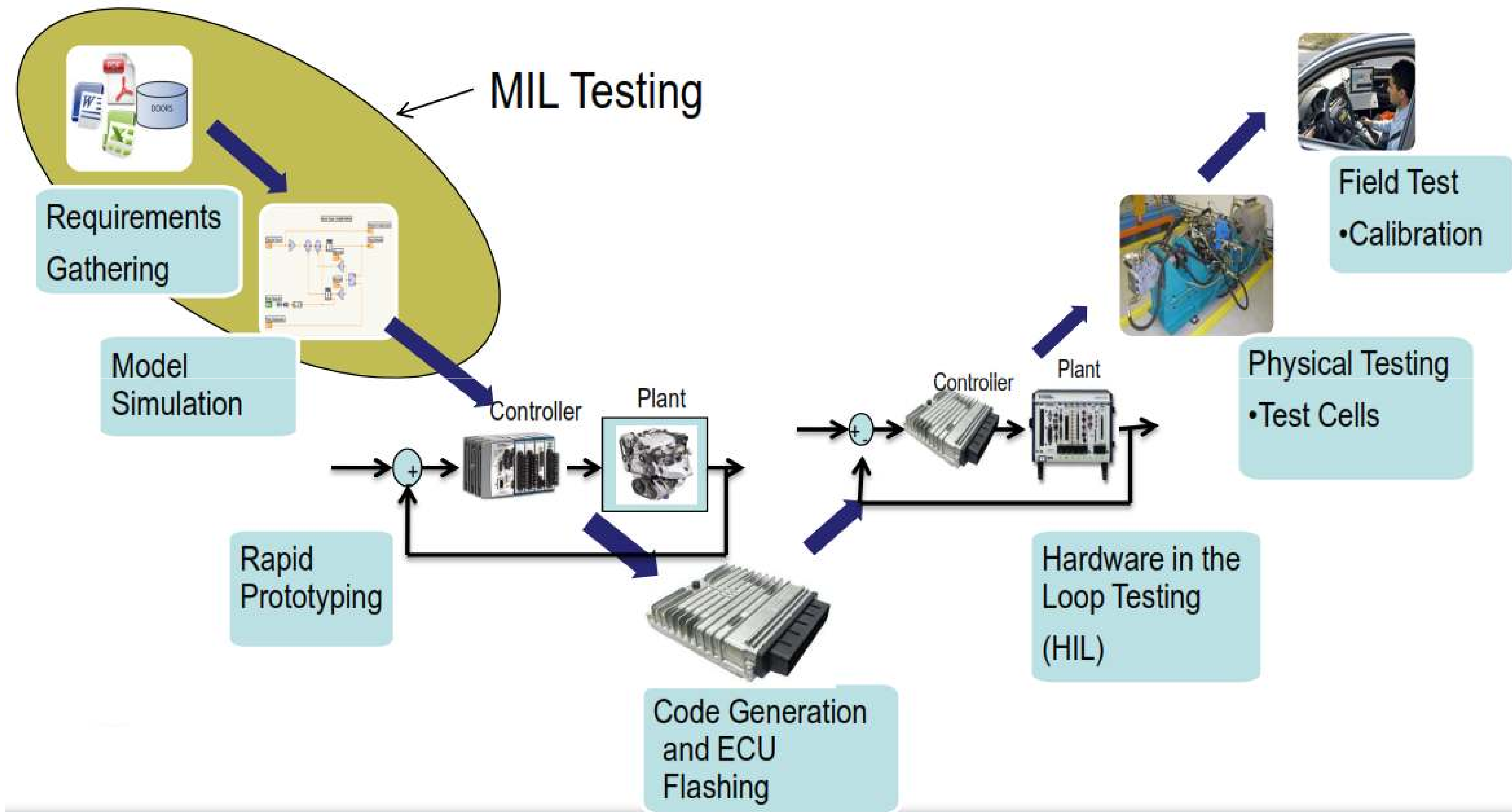


Rendszerteszt követelmények ISO26262

	Methods	ASIL			
		A	B	C	D
1a	Hardware-in-the-loop	+	+	++	++
1b	Electronic control unit network environments[a]	++	++	++	++
1c	Vehicles	++	++	++	++

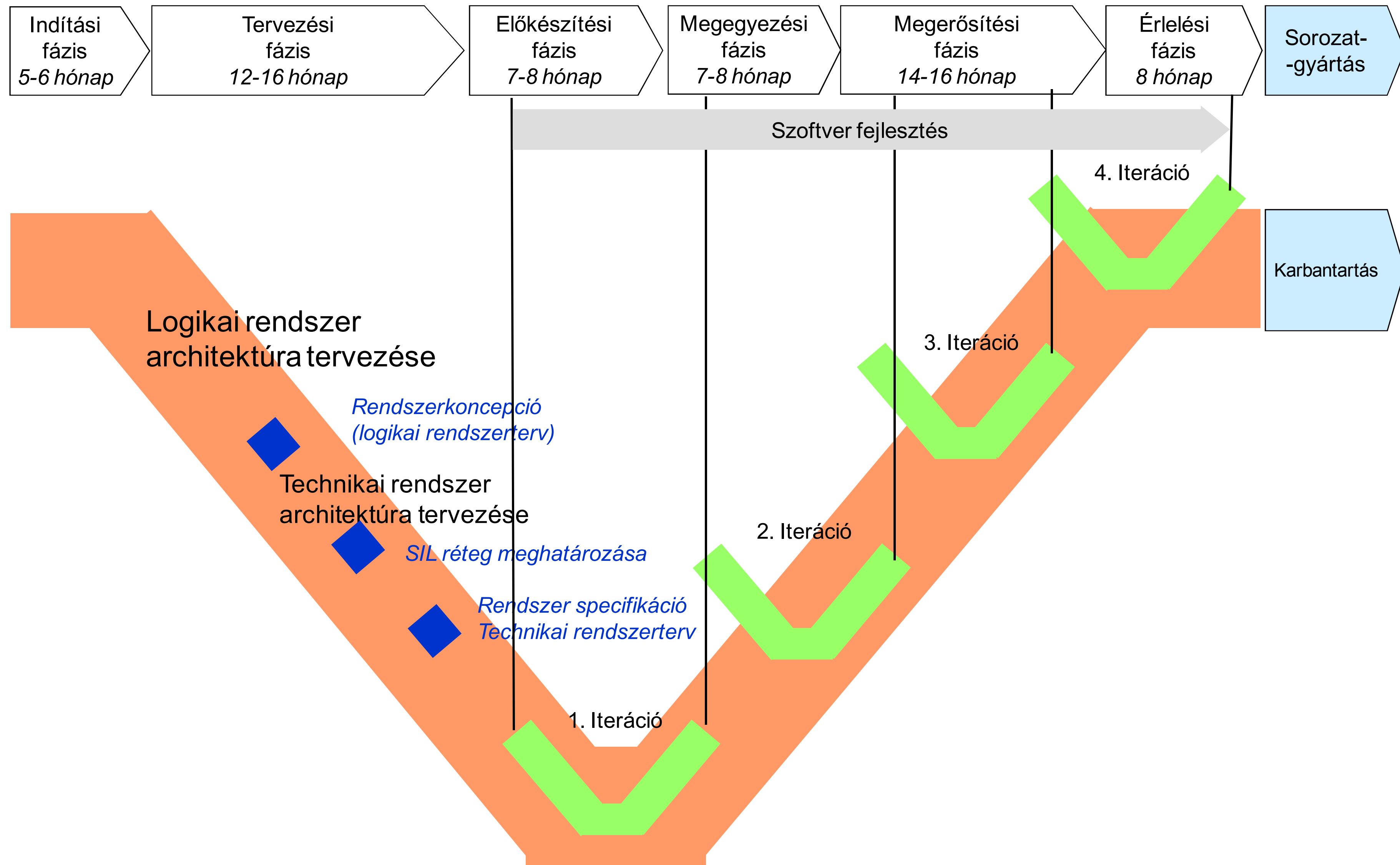
[a] Examples are "lab-cars", "rest of the bus" simulations or test benches partially or fully integrating the electrical systems of a vehicle.

MIL, SIL, HIL, Teszt cellák és kötődésük a V-modellhez

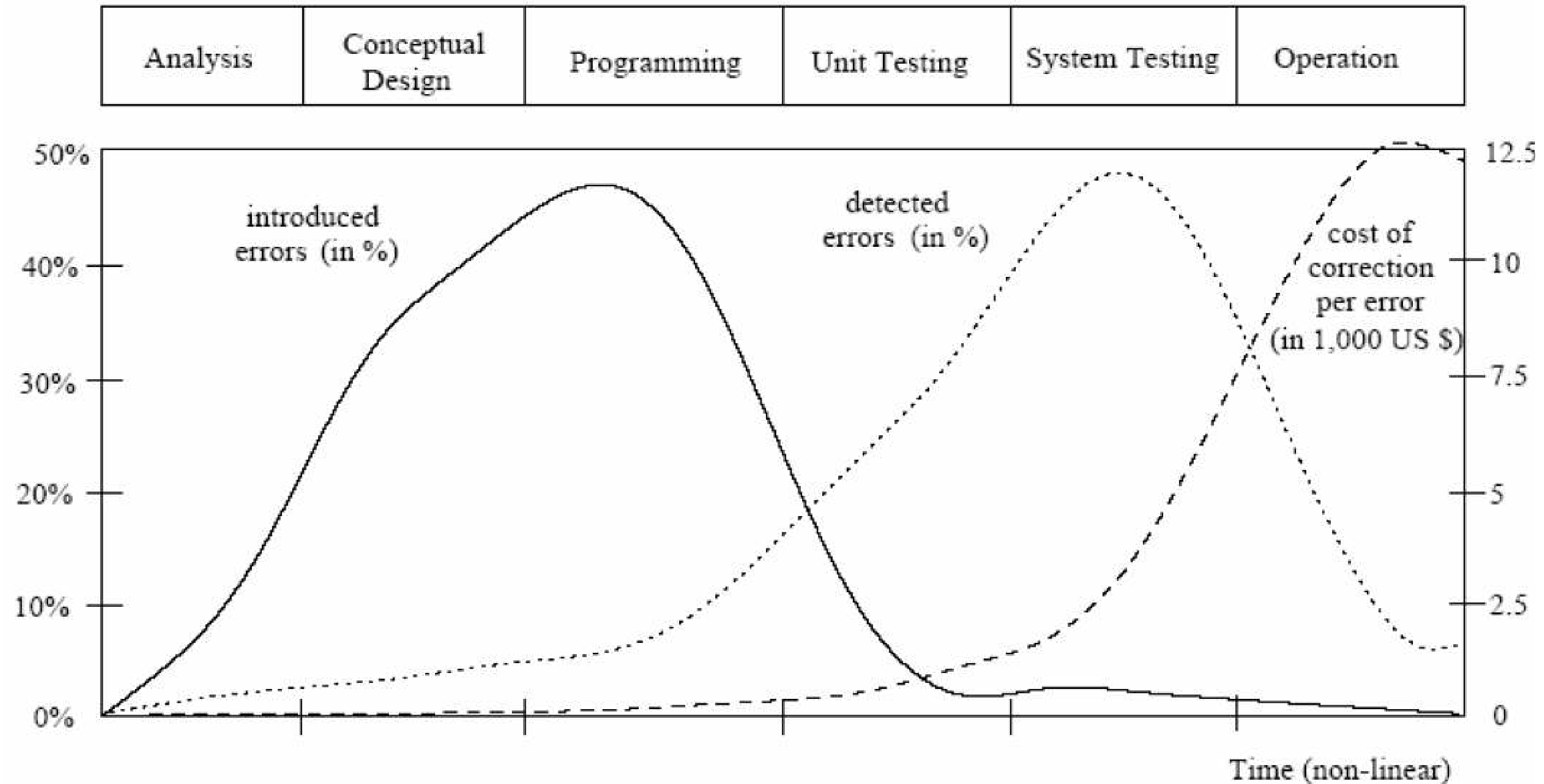


Forrás ni.com

V-modell a valóságban



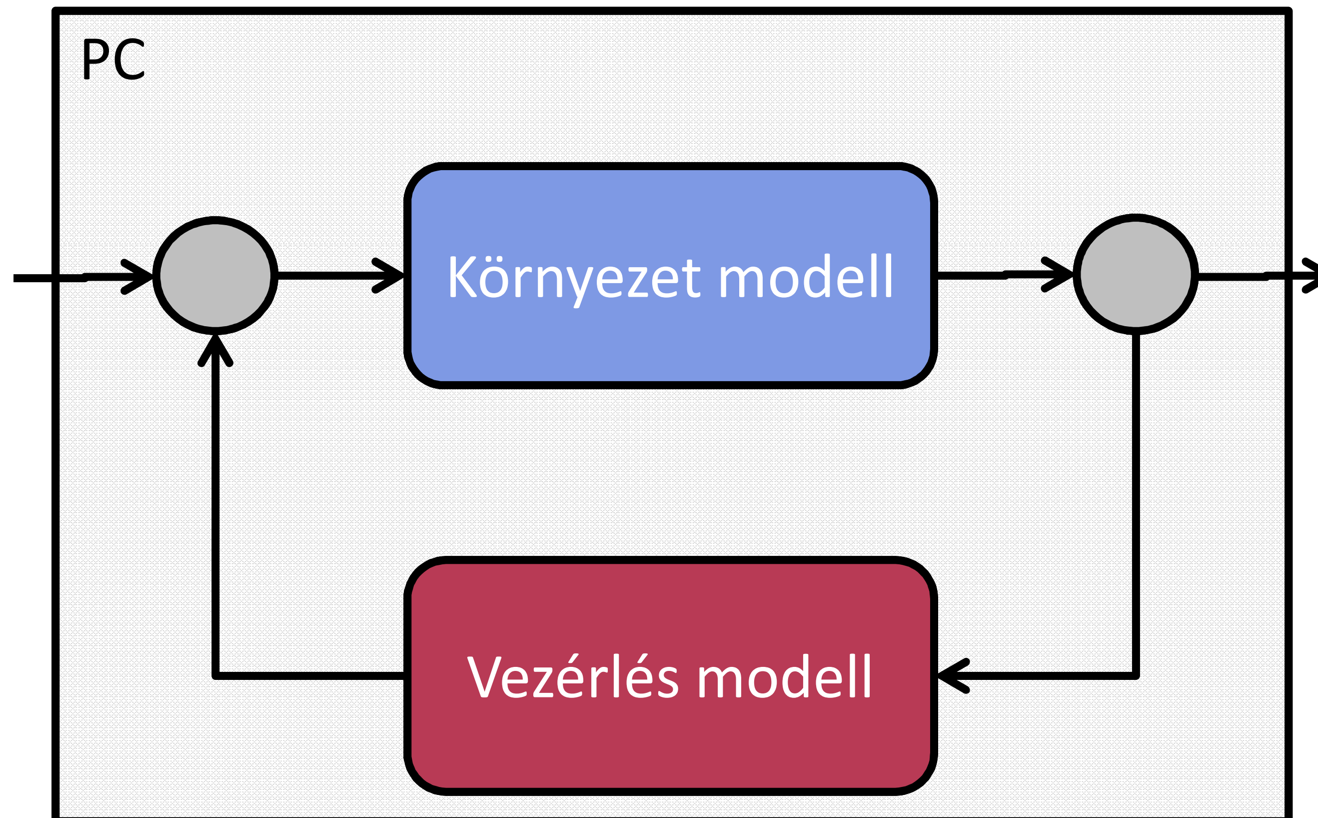
Miért fontos a korai és szisztematikus tesztelés?



From: P. Liggesmeyer et al., Qualitätssicherung Software-basierter technischer Systeme, Informatik Spektrum, 21:249-258, 1998. Quoted after J.P. Katoen, Principles of Model Checking, 2004/5. Copyright © by the authors.

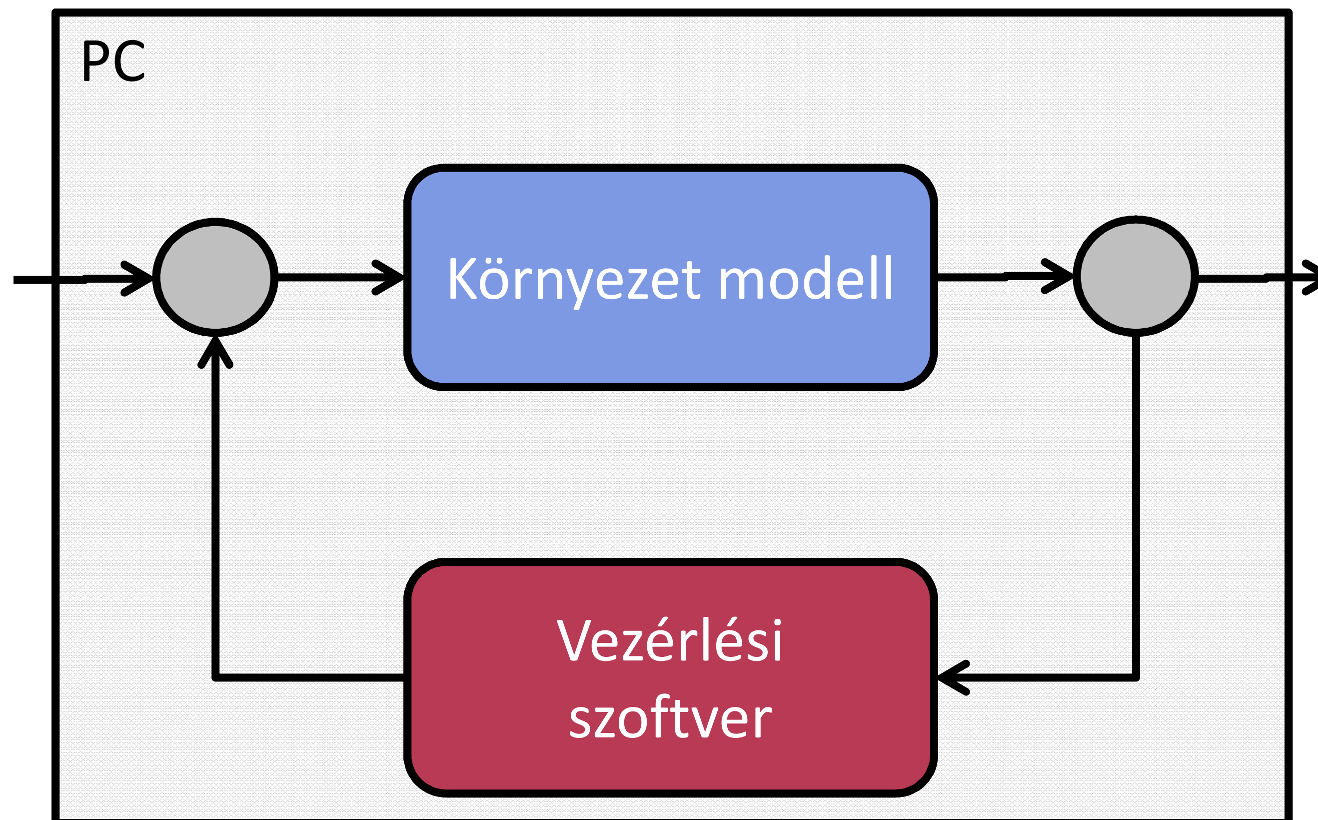
Model-In-the-Loop teszt

- Nagyon gyorsan a korai fázisban a fő funkció működési koncepciójáról kapunk visszacsatolást
  - Simulink, vagy LabVIEW



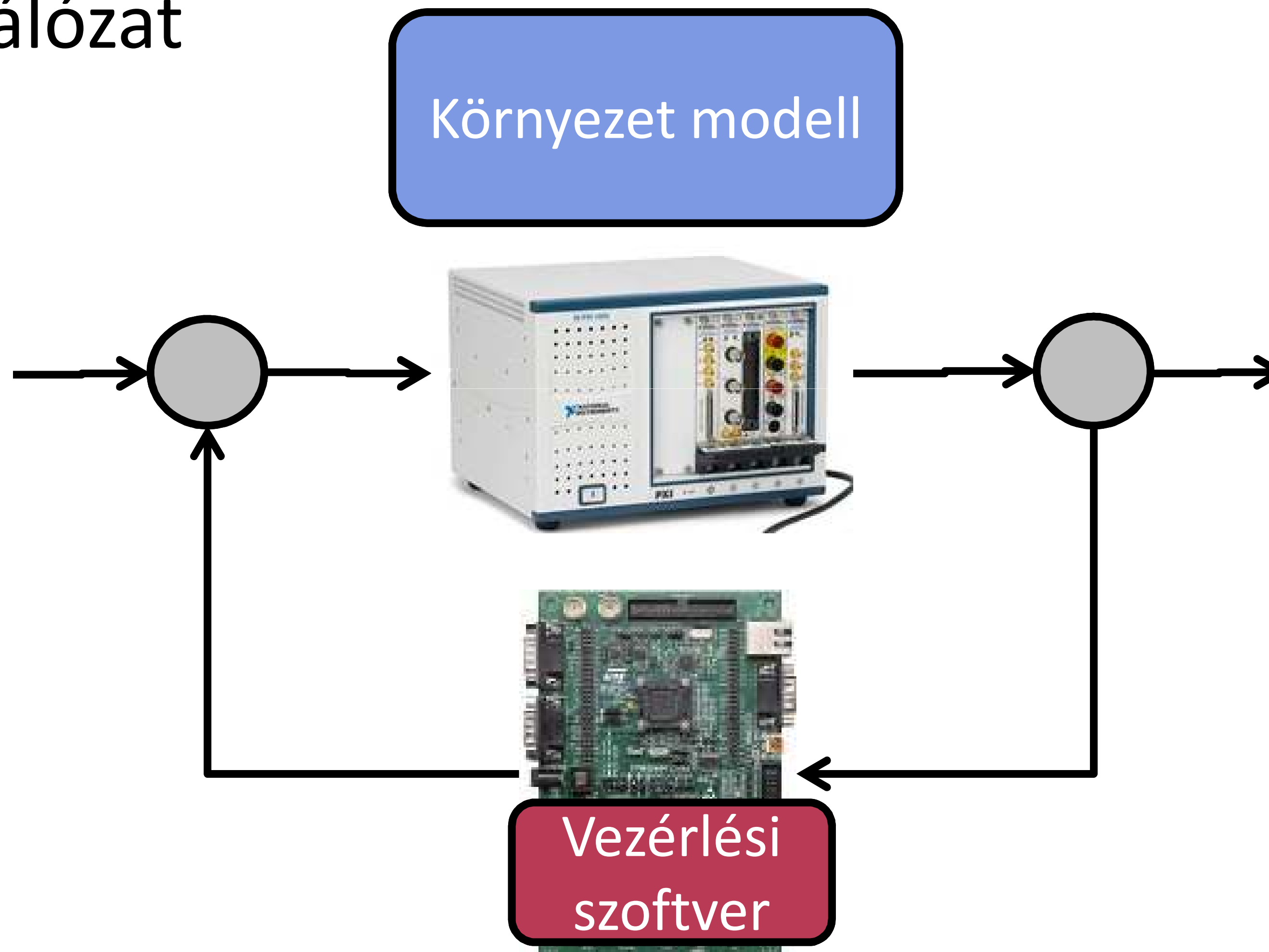
Software-In-the-Loop teszt

- A modellből generált kód
  - Különböző adatábrázolási problémák, generálási problémák kiderítésére



Hardware-In-the-Loop teszt

- Valós vezérlőn futó kód, valós interfészek
- hálózat



MIL, SIL, HIL összefoglaló táblázat

Teszt kategória	Vezérlő		Berendezés
	Hardver	Algoritmus	
Rapid Prototyping	gyors prototípus eszköz	modell	valós hardver
MIL	PC-s környezet	a teljes vezérlés modellje	modell
SIL	PC-s környezet	modellből generált kód	modell
HIL	valós hardver	a teljes vezérlés kódja	modell
Teszt cella	valós hardver	a teljes vezérlés kódja	valós hardver

Hardware-In-The-Loop tesztek

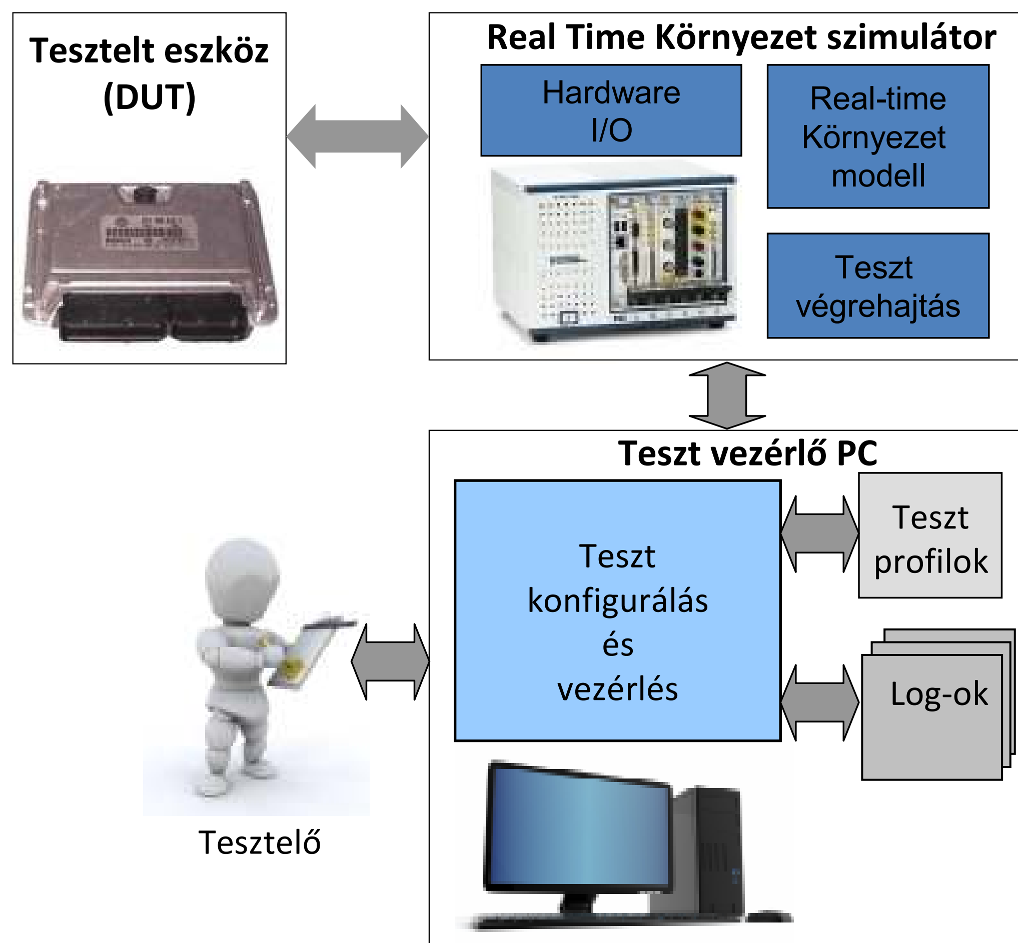
A beágyazott vezérlő egy a valóságot szimuláló környezetben hajtja végre a funkcionalitását

- Könnyű felműszerezés
- Megismételhető tesztek
- Nincs veszélyben a tesztelő a tesztelt eszköz és a környezet sem

Forrás: ni.com

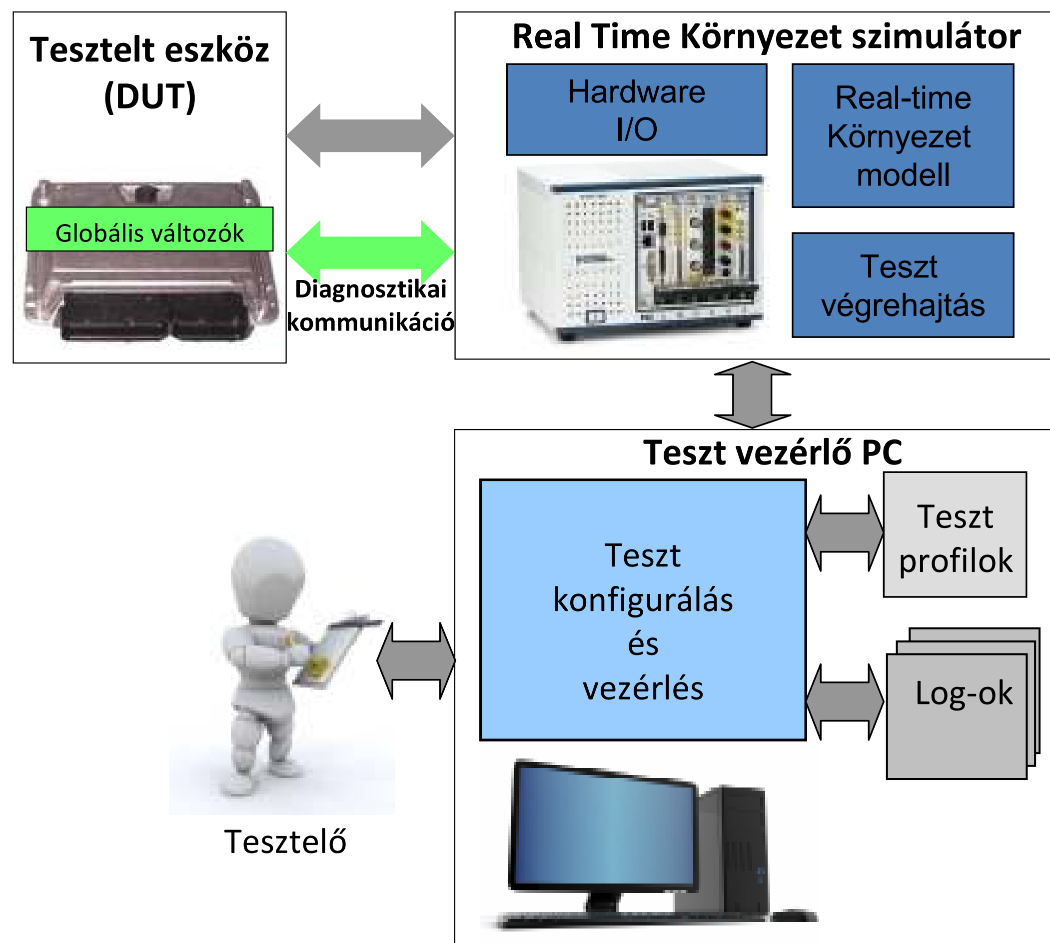
Egy tipikus HIL teszt környezet

- Teszt adatok
  - Analóg I/O
  - Digitális I/O
  - Kommunikációs



Egy tipikus HIL teszt környezet

- Teszt adatok
  - Analóg I/O
  - Digitális I/O
  - Kommunikációs
- Másodlagos információk
  - Rendszer belső működéséről
  - Globális változók



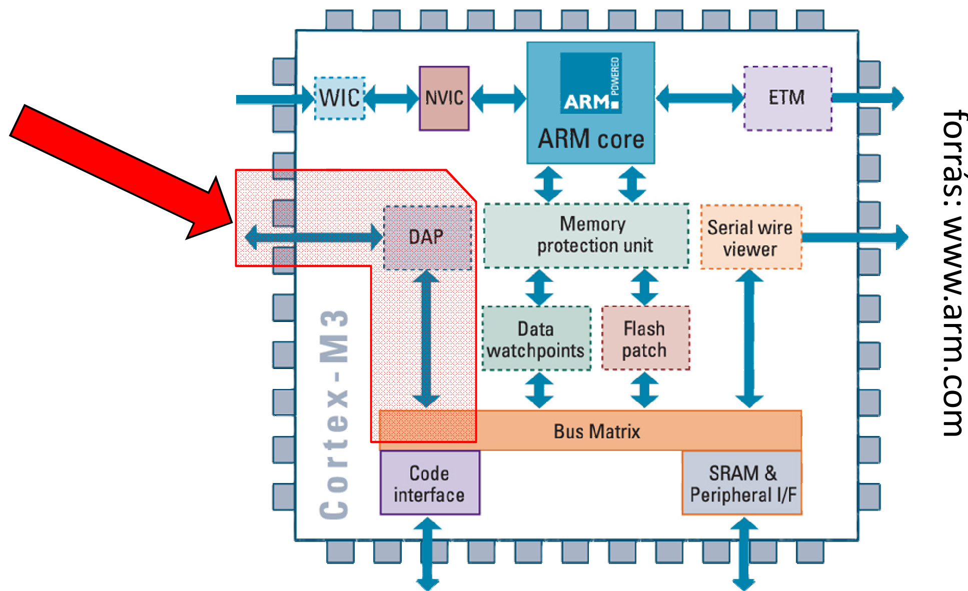
Másodlagos adatok kinyerése

- Diagnosztikai kommunikáció
  - Előnyök
    - Kész protokollok: CCP, XCP, UDS, KWP2000
    - Kész toolok: NI, Vector CANape, ETAS INCA
  - Hátrányok
    - Megosztott kommunikációs közeg
    - Limitált adatátviteli sebesség
    - Memória foglalás
    - Processzor erőforrás foglalás
- Hardware támogatott megoldások
- Szükségesek leíró file-ok, amik a globális változók tulajdonságait tárolják pl: A2L file (ASAM MCD-2 MC)

Modern mikrovezérlők debug képességei

- ARM magú vezérlők CoreSight szabvány alapú támogatás, más vezérlők IEEE-ISTO 5001-2003 (Nexus)
- Non-intrusive memória hozzáférés (CoreSight (AHB-AP), Nexus Class 3+)
  - Memória tartományok kiolvasása és írása futás közben

Debug hozzáférés a teljes belső buszmátrixhoz

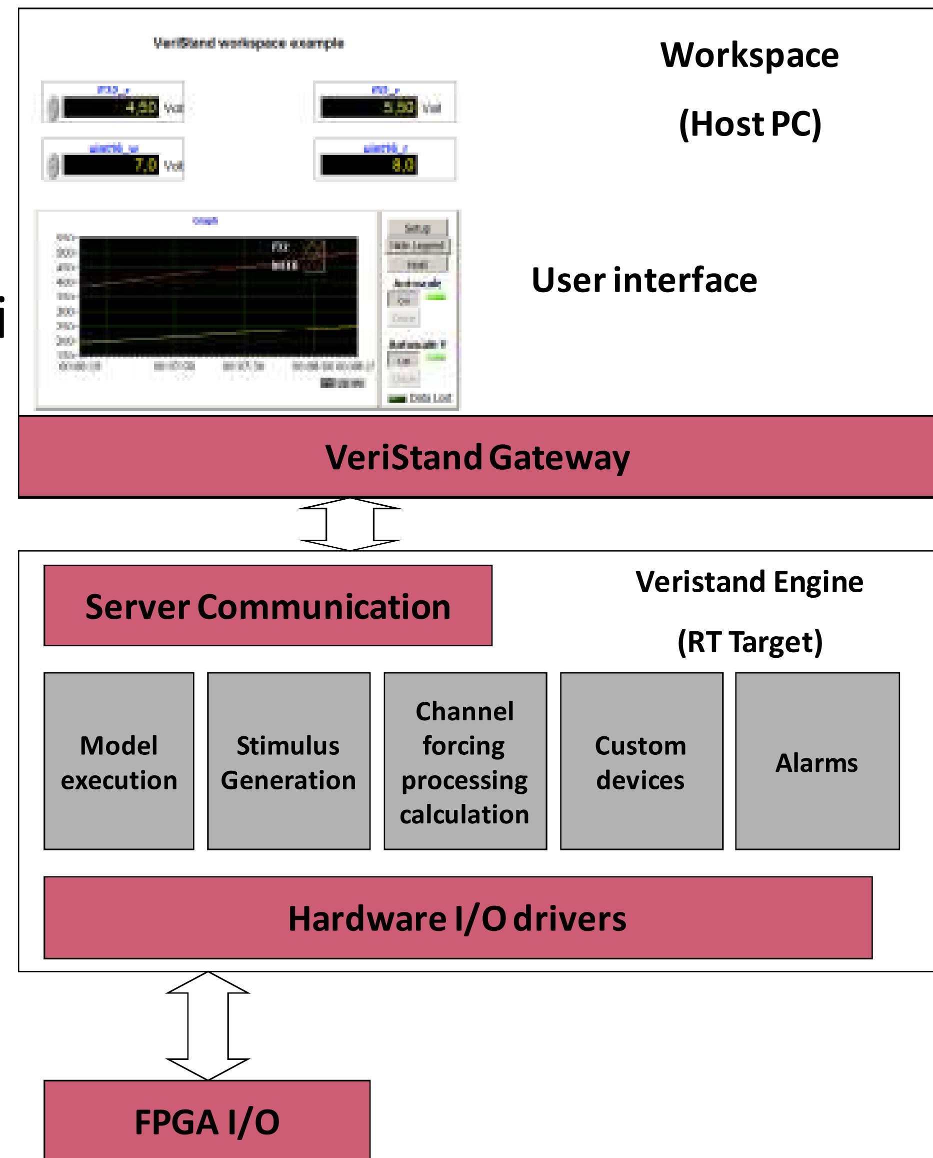


forrás: www.arm.com

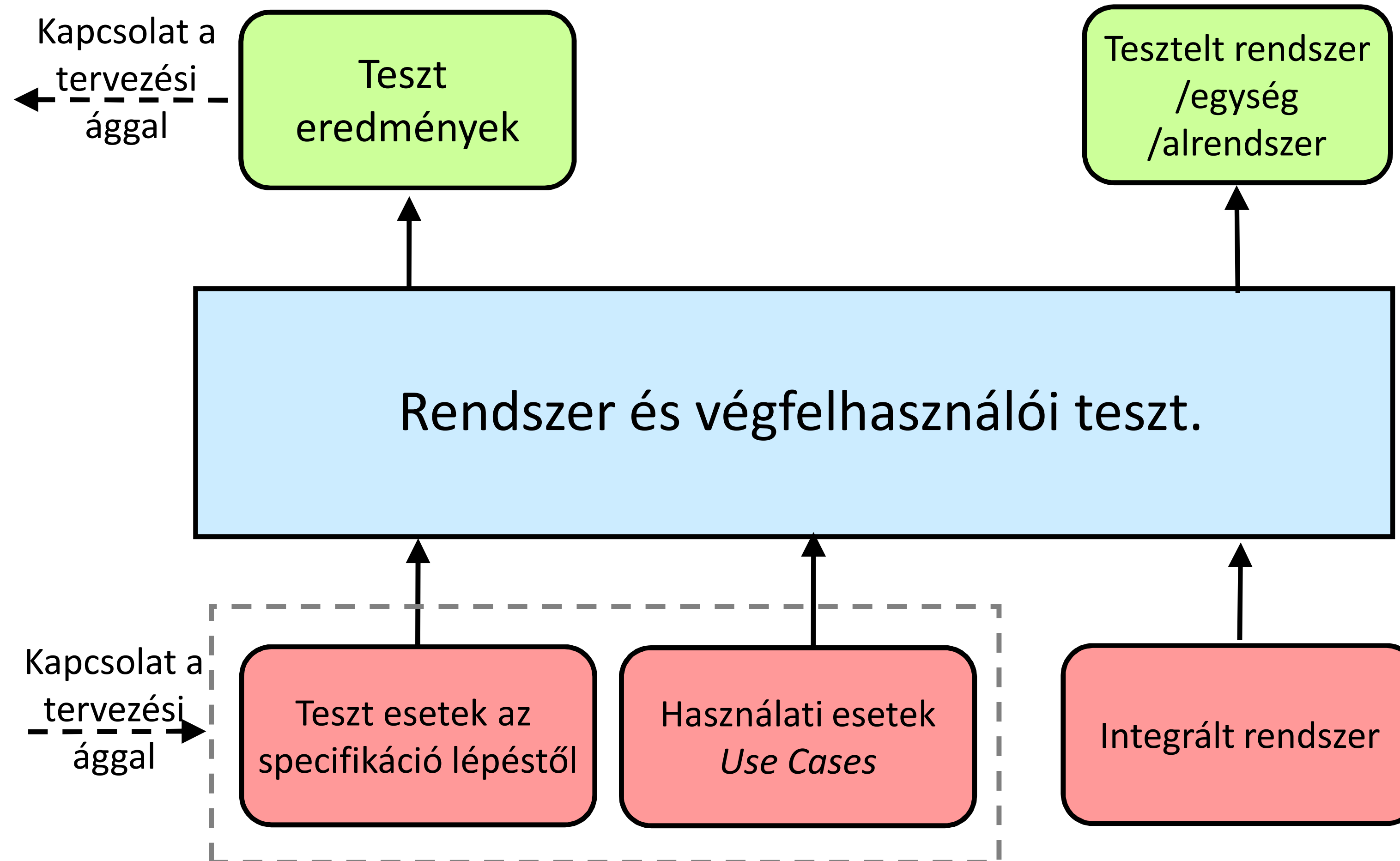
NI VeriStand

- Egyik legelterjedtebb HIL fejlesztő környezet
- Széles hardware támogatás
- Sokféle környezeti modell leírási opció
- Real-Time képesség

- *Szinte tetszőlegesen bővíthető*

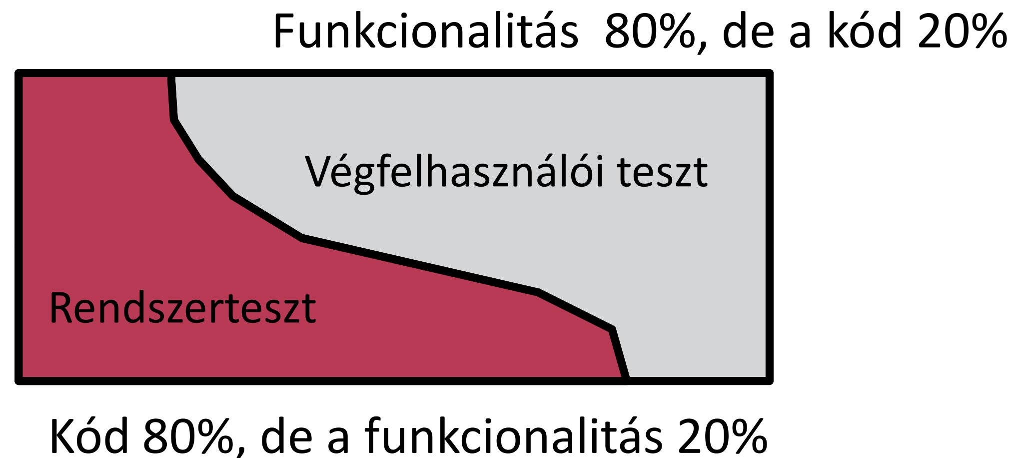


Végfelhasználói tesztek



Végfelhasználói tesztek

- A végfelhasználó bevonása a tesztelésbe
  - Ő tudja igazán mit szeretett volna
  - Más a fókusz



- Alfa teszt
  - A fejlesztő telephelyén fejlesztők bevonásával
- Béta teszt
  - A valós környezetben fejlesztők bevonása nélkül