1. A verifikáció és validáció technikáinak áttekintése: V&V feladatok a fejlesztési folyamat tipikus lépései során. Kritikus rendszerek jellegzetességei.

Kétféle **hiba** van: ami **fejlesztés közben** jön elő, és ami **élesítés után**, használat közben.

A fejlesztés közben előkerülő hibák (specifikáció hibái, tervezési hibák, implementációs hibák) kiküszöbölésére szolgának a *V&V technikák*. Az élesítés után előforduló hibák ellen (hardver hiba, konfigurációs hibák, műveleti hibák) *hibatűréssel* védekezhetünk, pl. redundáns erőforrásokkal. Minél hamarabb kiderül egy hiba, annál olcsóbb a javítása, ezért fontos a V&V.

Continuous Verification lényege: fejlesztési folyamatban végig ellenőrizve vannak a forráskódok. A fejlesztő egy featuret kódolási ajánlások betartásával írja, statikus analízis eszközök segítik munkáját, és unit tesztekkel ellenőrzi a működést. A kódot más fejlesztők reviewzák, és egy CI (continuous integration) eszköz minden featurre általánosan teszteli az egész alkalmazást, system test, integration test, e2e test. Így kerülnek productiönbe a fejlesztések.

Validációs és Verifikációs feladatok a fejlesztés során:

- Követelmények vizsgálata: Felhasználók, érdekeltek (stakeholder) elvárásai, víziói, kérései. Általuk validálható a rendszer. Meg kell tervezni, hogyan validáljuk a kész rendszert. *Checklist* összeállítása, miket kell ellenőrizni és hogyan, és *FMEA* (Failure Mode and Effect Analysis) *hibamód és hibahatás elemzése*, hiba lehetőség és hibahatás elemzése a hibák hatásának jelentőségének meghatározásához.
  Követelmény = egy feltétel vagy képesség melyre a felhasználónak szüksége van egy probléma megoldására, vagy egy cél elérésére (IEEE). Jellemzője, hogy *egyedi* és *megkülönböztethető* (ID), *konzisztens* (nincsenek ellentmondások), *egyértelmű* (nem lehet félreérteni), *verifikálható* (teszttel egyértelműen eldönthető, hogy teljesül-e a követelmény), *implementálható*. Gyors leírása a problémának, nem a megoldásnak!
- Rendszer specifikáció: A követelmények vizsgálatának eredménye. Lényegében a követelmények feladatokká alakítása a tervezők és fejlesztők számára. Szükséges specifikáció *review*. Szimulációk is végezhetők. Általa verifikálható a rendszer. Meg kell tervezni, hogyan teszteljük a kész rendszert.
- Rendszer tervezés: Hibafa, ETA (Enterprise technology architecture) infrastruktúra ajánlásainak megírása a rendszer tervezéséhez. Integrációs teszteket kell megtervezni.
- Modul tervezés: modellellenőrzés, gyors prototípusok. Modul tesztelést kell megtervezni.
- Modul implementáció: kódolási ajánlások, statikus analízis.
- Modul tesztelés: unit tesztek, kódlefedettség ellenőrzése
- Integrációs tesztelés: integrációs tesztek, modulok együtt működése. SIL/HIL/PIL: szoftver, hardver, és folyamat együttes tesztelése
- Rendszer tesztelés: Jól fejlesztettük-e le a rendszert? Objektív nézőpontok, automatizálható. Alapja a specifikáció. Nem szükséges, ha az implementációt közvetlenül a specifikációból generálták. Elfogadási teszt felhasználóktól/megrendelőktől/érdekeltektől.
- Rendszer validálás: Jó rendszert fejlesztettünk le? Szubjektív nézőpontok is. Alapja a megfogalmazott követelmények. Ha hiba van, az a követelmények hibájára vezethető vissza. Nem szükséges, ha a specifikáció nagyon egyszerű. Elfogadási teszt felhasználóktól/megrendelőktől/érdekeltektől.
- Support, karbantartás: hibák, megbízhatóság monitorozása

Reviewról általában: informális review, nincs formális folyamat, társ vagy vezető ellenőrzi. Átvizsgálás, mikor a review tárgyának írója vezeti végig az ellenőrző személyt az irományán, viszonylag informális. A technológiai reviewn kötött szabályok vannak, szakértők ellenőrzik, előzetes felkészülés szükséges a meetingre. Az inspekció, szemle egy szintén formális módszer, egy kiképzett moderátor vezeti.

Kritikus rendszerek: *Nyomon követhetőség* kiemelten fontos a követelmények esetén. *Előrefelé* nyomon követhető, ha minden követelmény hivatkozik egy kódrészletre, ami teljesíti a követelményt. Bizonyságot ad arra, hogy minden implementálva van. *Visszafelé* nyomon következő, ha az implementáció minden sorához hivatkozva van egy vagy több követelmény, ami miatt szükség van a kódra. Így elkerülhetőek a felesleges funkciók implementálása.

Biztonság = a feltételezése annak, hogy a rendszer adott körülmények között nem vezet olyan állapothoz, ahol emberi élet, egészség, tárgy vagy a környezet kárt szenvedne. Ennek biztosítói a *tanúsítványok*, melyek alapjai a *szabványok*.

Egy **biztonsági funkció** célja elérni vagy megtartani a *biztonságos állapotot*.

A **biztonságintegritás** adja meg a *valószínűségét* annak, hogy a biztonságkritikus rendszer megfelelően végre tudja hajtani az elvárt biztonsági funkciókat minden meghatározott feltétel mellett egy adott időintervallumban. Ennek szintjei a **SIL** (Safety Integrity Level), amit a tolerálható hiba ráta (THR) értékek alapján osztottak be

SIL	Probability of dangerous failure per hour per safety function
1	$10^{-6} \leq THR < 10^{-5}$
2	$10^{-7} \leq THR < 10^{-6}$
3	$10^{-8} \leq THR < 10^{-7}$
4	$10^{-9} \leq THR < 10^{-8}$

2 Forráskód ellenőrzés: A statikus analízis eszközök típusai. Ellenőrzés által detektált tipikus hibák. Ellenőrzési módszerek és tulajdonságaik. Az absztrakt interpretáció alapelvei.

Statikus analízis eszközök: statikus verifikációs technikák, automatikus módszerek a kód futás idejű tulajdonságainak futtatás nélkül való vizsgálatára. Céljuk olyan problémák detektálása, amiket nem ellenőriznek a hagyományos fordítók, és tesztelés során is nehéz megtalálni őket.
Az összes futási idejű tulajdonság ellenőrzése lehetetlen, különböző megközelítéssel élnek az analízis eszközök, a elemzésre szánt idő és a precizitás között keresnek kompromisszumos megoldást.

- *Folyam érzékeny – Flow sensitive*: programkódok végrehajtási sorrendjét ellenőrzi. Pl. két pointer ugyanarra a memóriaterületre mutat az i. kódsor után.
- *Útvonal érzékeny – Path sensitive*: különbséget tesz különböző lefutási lehetőségek között. Csak a megvalósítható lefutásokat veszi figyelembe. Pl. más eszközzel ellentétben nem jelzi egy osztásnál, hogy talán nullával osztás, ha az előtte lévő sorokból látszik, hogy biztos nem az.
- *környezet érzékeny – Context sensitive*: függvények mindig máshogy vannak vizsgálva, attól függően, hol hívják őket. Pl. nem jelzi a függvényben lévő osztásnál, hogy talán nullával osztás, ha látja, hogy nem nullával hívták meg.
- *Eljárások közötti – Interprocedural*: megvizsgálja a függvény törzsét minden hívásnál. Pl. nem jelzi, hogy talán null pointert adott vissza a függvény, ha látja, hogy az sosem tud vissza adni null pointert.

Detektált hibák:

- Nem megfelelő erőforrás kezelés => memory leak
- Illegális műveletek (nullával osztás, overflow, túlindexelés, null pointer)
- Halott kód, ami sosem fut le, és használatlan változó
- Félkész kód (inicializálatlan változó, elmaradt visszatérési érték, hiányzó case switch-ben)
- Egyéb: nem terminál, nem lekezelt kivétel, versenyhelyzet

Ellenőrzési módszerek:

- *Mintaillesztés-alapú*: általában abstract syntaxt tree-n (AST) alapul, NEM útvonal és kontextus érzékeny. Jó: nagy projektek vizsgálatára is jó, teljesen automatizált. Rossz: egyszerű tulajdonságok (minták) vizsgálatát tudja csak, sok false pozitívot és false negatívot eredményez.
- *Típus ellenőrző*: típusos programnyelveknél. Elméletileg bármilyen tulajdonság vizsgálatára alkalmas. A gyakorlatban csak egyszerűbb tulajdonságokat vizsgál. Általában a fordítóban van alapból, teljesen automatizált, hatékonyan eldönthető tulajdonságokat vizsgál.
- *Absztrakt interpretáció*: végigvezeti az információkat a programon. A változók értékkészletének szűkítésével próbálja meghatározni, milyen értékek esetén történik hiba. Folyamat és útvonal érzékeny, kontextus érzékeny is és eljárások közötti is. Nagyrészt automatizált, de néha felhasználói beállítást igényel. Hibákat mindig megtalálja, de false alarm is lehetséges.
  Alapelvek: konkrét domént *absztrakt doménre képezi le*, változókat valamilyen absztrakciós függvény segítségével, pl. változó értéktartománya, konkrét utasításokat meg valamilyen absztrakciós műveletekre, pl. tartományok uniója. Analízist az absztrakt domén segítségével végezzük. Ez egy közelítő megoldás, ha az absztrakt és konkrét domén közötti leképezés betart bizonyos szabályokat, akkor a konkrét fixpontok biztonságos megfelelőjét kapjuk: az absztrakt domén lefedi a konkrét domént (érték nem veszik el), de megjelenhetnek az absztrakt doménben olyan értékek, amik a konkrét esetben nem

fordulhatnak elő.

Nem-relációs domének esetén változók közötti relációk nem őrizhetők meg (pl. x<y elveszik). Nem-relációs absztrakció az *előjelek, intervallumok, paritás és kongruenciák* (egyenlőtlenség kimutatható kongruenciákkal, látjuk pl. ha nem nullával osztás lesz az 1/(x-y)) szerinti absztrakció.

Relációs domének a *Difference Bound Matrices* (DBM) ahol **x-y≤c**, **x≤c** és **-x≤c** formájú egyenlőtlenségek konjunkciója jelöli ki a doméneket (valósidejű rendszerek ellenőrzéséhez használják, órák különbségeihez), az *Octagon* **ax+by≤c**, alakú egyenlőtlenségek alapján zárja intervallumba az értékeket, ahol a és b -1,0,1 lehet, több mint két változóra az *Octahedra* domén ad absztrakciót, a *Polyhedra* pedig $\mathbf{a_1x_1+\ldots+a_nx_n \le c}$ alakú egyenlőtlenségek alapján közelít.

- *Modellellenőrző*: formális nyelven megadva a program működését, minden lehetséges állapoton és állapotátmeneten végig iterálva ellenőrzi a végrehajtást. Folyamat és útvonal érzékeny, néhány eszköz kontextus érzékeny is és interprocedural is. Általában automatizált, de szükség lehet információra a felhasználótól. Teljesen formális, nincsenek false alarmok és észre nem vett bugok. Skálázódása azonban limitált, nagy modellekre nem hatékony. Szimbolikus technikákkal, korlátos modellellenőrzéssel, absztrakcióval lehet javítani rajta.
- *Tétel bizonyító*: logikai formulákkal megadva a programot és tulajdonságokat, alkalmazza a következtetési szabályokat hogy bebizonyítsa, hogy a programra teljesül egy tulajdonság. Teljesen formális, nincsenek false alarmok, se észre nem vett bugok. Felhasználói közreműködés szinte mindig szükséges. Hátránya, hogy a formális leírás megadása szakértelmet igényel.

Method	**Flow sensitive**	**Path sensitive**	**Context sensitive**	**Interprocedural**	**Arbitrary property**	**Automated**	**No false alarms**	**No missed bugs**	**Large codebase**
Pattern matching	X	X	X	X	X	✓	X	X	✓
Type checking	X	X	X	X	X	✓	(✓)	X	✓
Abstract interpretation	✓	(✓)	(X)	(X)	(✓)	(✓)	X	✓	(✓)
Model checking	✓	✓	(✓)	(✓)	✓	(✓)	✓	✓	(X)
Theorem proving	✓	✓	(✓)	(✓)	✓	(X)	✓	✓	X

3 Szoftver tesztelés alapjai: Tesztelés definíciója és céljai. Tesztelési alapfogalmak. Tesztelés folyamata, szintjei és típusai. Teszt orákulumok típusai.

Definíció:

- Egy tevékenység a termék minőségének kiértékeléséhez és javításához, hibák és problémák találásával.
- Egy tevékenység amely során a rendszert vagy komponenst meghatározott körülmények között futtatják, az eredményeket megfigyelik és rögzítik, és adott szempontok alapján értékelik a rendszert vagy komponenst.
- A folyamat, ami tartalmazza az összes életciklus tevékenységeit, statikusakat és dinamikusakat is, foglalkozik a tervezéssel, előkészítéssel és kiértékeléssel a szoftver termék és kapcsolódó munkák során. Azért, hogy megállapítsa, megfelel-e a követelményeknek, hogy demonstrálja, hogy valóban el tudja végezni a célzott feladatot, és hogy megtalálja a hibákat.
- A folyamata egy termék kiértékelésének azáltal, hogy felfedezésszerűen tanuljuk a használatát, vagyis kérdezünk, kutatunk, modellezünk, megfigyelünk és következtetünk, kimeneteket ellenőrzünk, stb.

Célok:

- Információ a minőségről
- Döntés hozás segítés (pl. release időpontja)
- Hibák detektálása
- Hibák megelőzése

Alapfogalmak: *teszt bemenet* **->** SUT **->** *teszt kimenet* **->** Orákulum **->** *eredmény*

- Teszteset (test case): bemeneti értékek és végrehajtási előfeltételek, várt eredmények és végrehajtási utófeltételek halmaza
- Tesztkészlet (test suite): tesztesetek halmaza
- Orákulum (test oracle): várt eredmények származtatása, összehasonlítása
- Eredmény (verdict): Sikeres (pass), sikertelen (fail), nem meggyőző (inconclusive), hiba (error)

Folyamata:

- Tervkészítés (plan): teszt szkópja, kockázatok, célok, stratégia, szükséges erőforrások (pl. emberek, tesztkörnyezet), többi lépés beütemezése, végcél meghatározása (pl. lefedettség). A kész terv tartalmazza a célokat, tesztelendő objektumokat és tesztkörnyezeteket, erőforrásokat és szerepeket, és ütemtervet.
  Kontrol: monitorozás, szükség esetén módosítás a terven
- Elemzés és tesztesetek tervezése (design): mit kell és lehet tesztelni? Tesztesetek megtervezése és specifikálása: cél, előfeltételek, lépések, tesztadatok, elvárt eredmény, ellenőrizendő adatok. Azelőtt kell ezeket tisztázni, hogy megírnánk a teszt kódját.
- Implementálás és végrehajtás: manuálisan vagy automatikusan is lehet tesztelni, de nem mindent lehet, vagy nem mindent éri meg automatizálni. Végrehajtjuk a teszteket, rögzítjük az időt, környezetet, teszt alatt lévő rendszer verzióját, kimeneteket. Feljegyezzük a problémákat.
- Kiértékelés és rögzítés: cél elérése után (pl. tesztlefedettség), kiértékelés, összegzés írása
- Teszt lezárás: az összes fő mérföldkő után összegyűjtjük a tapasztalatokat, visszajelzéseket, lezárjuk a folyamatot és elmentjük a dolgokat, amik később újra felhasználhatóak (eszközök, környezetek)

Szintjei: unit/module, integrációs, rendszer, elfogadási, alpha és béta

Típusai: funkcionális, nem-funkcionális, regressziós (<- változtatások után, hogy az eddigi működések még mindig jók e, csak néhány általános teszt)

- Kísérletezés alapú: ad hoc, felfedező jellegű, a tesztelőnek nagy szabadsága és felelőssége van. Teszt tervezése, végrehajtása és értelmezése párhuzamosan történik.

- Specifikáció alapú: cél a specifikációban leírtakat alátámasztani, hogy úgy működik ahogy kell. A tesztelt rendszer black box, nem ismert a belső felépítése, csak azt elvárt funkcionalitás
- Struktúra alapú: a tesztelt rendszer white box, ismert a belső felépítése, és az alapján tesztelik.
- Hiba alapú: keressük a hibalehetőségeket, az előző hibák alapján és tipikus hibalehetőségekből kiindulva. Mutációs teszt is csinálható, módosítjuk a kódot, kiértékeljük a tesztet, módosítjuk a tesztet, új teszteket hozunk létre.
- Valószínűségi: Determinisztikus vagy valószínűségi módon származtatjuk a teszteseteket, választhatunk véletlenszerű, működési vagy statisztikai módszert hozzá.

Teszt orákulumok típusai: a tesztorákulum olyan elvek és mechanizmusok gyűjtőeszköze, amik segítenek eldönteni hogy egy program átment e a teszten. Döntés lehet: sikeres, sikertelen, hibára futott, nem meggyőző.

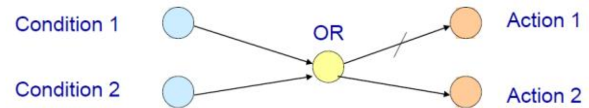
Kihívások: nem tesztelhető programok, amelyeket azért írtak, hogy választ adjanak egy kérdésre. Vagy olyan esetek, amikor átmegy a teszten, de mégsem jó, mert túl lassú, nem használ autentikációt, használhatatlan UI. Vagy épp ellenkezően, nem megy át a teszten, pedig jól működik, mert nem gondoltunk néhány különleges esetre.

- Specifikált: emberek, szöveges specifikáció alapján, modelleket használva (UML, FSM)
- Implicit: kivételek, összeomlás, biztonsági hibák tesztelése, robosztusság tesztelése, teljesítménytesztek
- Származtatott: előző program verzióhoz mérve, regressziós teszt, csak a különbségeket ellenőrzik. Vagy különböző implementációk (N-verziójú programozás) tesztelése. Vagy feltevések ellenőrzése, validálás, invertálható funkciók

4 Specifikáció alapú teszttervezés: Specifikáció alapú módszerek áttekintése. Döntési táblák. Kombinatorikus módszerek, n-wise testing.

Döntési vagy ok-okozat analízis: bemenetek és kimenetek összefüggései (pl. ár számítás, autentikáció). Feltételek és okozatok vizsgálata (bemeneti paraméterek), vagy akciók és hatásuk vizsgálata (kimeneti paraméterek). Reprezentáció ok-okozat gráfként vagy döntési táblaként.

- <u>Ok-okozati gráf</u>: bementi és kimeneti paraméterek összekapcsolása OR AND és NOT csomópontokkal. Tesztek tervezéséhez használják, minden útvonal lefedését segíti. Igazságtáblákat is könnyű készíteni a segítségével. Eredetileg hardver teszteléshez találták ki.



- <u>Döntési tábla</u>: minden feltételt vagy akciót boolean értékkel jelöl. Feltételek/akciók a sorokban, üzleti szabályok az oszlopokban, ezek lesznek a tesztesetek.

	Rule 1	Rule 2	Rule N
Conditions			
Condition 1	T	T	
Condition 2	F	T	
...			
Actions			
Action 1	X		
Action 2		X	
....			

Kombinatorikus tesztelés: Ha túl sok bemeneti paraméter van, az összes lehetőség kipróbálása túl sok tesztesetet eredményezne, fontos azonban a tesztelés, mivel konkrét, akár ritkán használt kombinációk is okozhatnak hibát. Tesztelhetjük őket *ad hoc* módon, intuitívan, követelmények alapján, tipikus hibalehetőségekre gondolva. Vagy arra figyelve, hogy *minden bemeneti paramétert legalább az egyik* teszteset kipróbáljon. Ez nem annyira jó, mert kimaradhatnak olyan paraméter kombinációk, amiket fontos lenne tesztelni. Legjobb az *n-szeres* tesztelés, minden tetszőleges n paraméterre az összes lehetőség kipróbálása. Ha n=2, az a *páronkénti* tesztelés.

5 Struktúra alapú teszttervezés: Struktúra alapú módszerek áttekintése. Vezérlési folyam alapú kritériumok alapfogalmai. Feltétel, C/DC és MC/DC lefedettségek.

A belső struktúra modellek esetén a modell struktúráját jelenti, kód esetén a kód felépítését, például vezérlési folyam gráffal felírva. Vannak **utasítások**, **blokkok** (egymás utáni utasítások, elágazás nélkül), **feltételek** (logikai kifejezés logikai operátor nélkül), **döntések** (több feltétel összekötése logikai operátorral), és **elágazások**, **ágak**. Egy **útvonal** események, például végrehajtható utasítások egy sora, tipikusan egy komponens esetén egy **kezdőponttól** egy **végpontig**.
A **vezérlési folyam gráf** csúcsai blokkok, utasítások egy sorozata egyetlen belépési és kilépési ponttal. Az élek közöttük jelentik, hogy az egyik blokkból el lehet jutni a másikba a kód futása során.

Kritériumok:

- Utasítás lefedettség: tesztek által lefedett utasítások / összes utasítás. Vagyis összes csúcs le legyen fedve a gráfban. Nem fed le minden ágat.
- Döntési lefedettség: elágazások lefedettsége tesztek által / összes elágazás. Minden ágat jó lefedni (élek a gráfban). Minden állapotot is lefed. Nem biztos, hogy minden döntési kombinációt érint.
- Feltétel lefedettség: tesztel által lefedett kombinációi a feltételeknek / összes lehetséges kombináció. Elv: minden feltételnek egyszer ki kell értékelődnie hamisnak és egyszer igaznak a tesztek során! Nem biztosít 100% döntési lefedettséget.
- C/DC, feltétel/döntés lefedettség: e kettő keveréke. Minden döntést és minden feltétel minden lehetséges értékét kipróbálja. Nem veszi figyelembe, hogy egy feltételnek van-e jelentősége a döntésben (pl. true || akármi esete)
- MC/DC, módosított C/DC: minden bemeneti és kimeneti pont legalább egyszer meg legyen hívva. Minden feltétel minden döntésben vegyen fel minden lehetséges értéket legalább egyszer. Egy döntésben minden feltétel meg van vizsgálva úgy, hogy tényleg hatása van a végeredményre.
- Több feltétel szerinti lefedettség: Miiinden kombinációt ki akarunk próbálni. N feltétel esetén 2^N teszteset is szükséges lehet, lusta kiértékeléssel kicsit kevesebb. Nem mindig praktikus.

	Utasítás	Döntés	Feltétel	C/DC	MC/DC	Több
Minden bemeneti és kimeneti pont		✓	✓	✓	✓	✓
Minden utasítás	✓					
Minden döntés, összes lehetőséggel		✓		✓	✓	✓
Minden feltétel minden döntésben minden lehetőséggel			✓	✓	✓	✓
Minden feltétel minden döntésben úgy, hogy befolyással van a végeredményre					✓	✓
Minden lehetséges kombinációi a feltételeknek						✓

- Fő útvonal lefedettség: lefedett független útvonalak / összes független útvonal. 100%-os útvonal lefedettség biztosítja a full utasítás, döntés, és több feltétel szerinti lefedettséget! Nagyon nem praktikus, ha ciklus is van a fában.
- Egyéb: ciklusok esetén 0,1 vagy több iteráció futtatása, versenyhelyzet esetén többszálú tesztelés, stb.

Hasznos: program nem tesztelt részeinek megtalálása, tesztkészlet teljességének ellenőrzése, használhatjuk célként is, hogy mekkora lefedettséget akarunk elérni, és teszt generálásra is.
Nem használható: hiányzó vagy nem implementál t követelmények keresésére, tesztelésére, kód minőségének ellenőrzésére.

6 Kód alapú tesztgenerálás: Kód alapú tesztgenerálás célja és korlátai. Módszerek: szimbolikus végrehajtás, véletlen generálás, annotáció alapú és keresés alapú generálás. Tipikus kihívások és eszközök.

Célok: olyan esetben lehet nagyon hasznos, amikor egy eddig alig tesztelt szoftvert kell tesztelni, és adott a forráskód vagy a lefordított program. A költséges emberi tesztelők helyett generálhatunk is teszteket!

Korlátok: Elvárt kimenetet nem tudjuk, de tesztelhetünk errorokat, assert feltételeket a kódban különböző bemenetekkel, manuálisan beletett assertekkel még hatékonyabbá tehető. Már megkapott kimeneteket újra felhasználhatjuk pl. regressziós tesztelésnél.

Módszerek:

- Szimbolikus végrehajtás: nézzük kód vezérlési folyam gráfját. 70-es években: szimbolikus változók az eredetik helyett, kifejezést alkotunk belőlük egy útvonalon a fában, és kényszer megoldóval keresnek egy megoldást, ami az adott útvonalhoz tartozó bemeneteket adja, ezzel teszteljük. Manapság már több számítási kapacitással rendelkezünk, és új módszereket dolgoztak ki hozzá.
  Kibővített statikus szimbolikus végrehajtás: eredeti módszer nem működik túl hosszú útvonalakra (túl sok kényszer miatt), és nem tudja eldönteni, egy útvonal tényleg kiértékelhető-e. Módosítás: szimbolikus és konkrét végrehajtás ötvözése: dinamikus szimbolikus végrehajtás (DSE). A kényszereket lépésenként kiértékelve, monitorozva, lépésenként választ útvonalat, amíg tud.
- Véletlen generálás: lehetséges bemenetek közül véletlenszerűen választva. Nagyon gyors és nagyon olcsó. Ha nem talál hibát, különböző részeit nézi a domainnek. Kiválasztás különbözőség, távolság stb alapján. A Randoop eszköz heurisztikák segítségével képes kiszűrni az invalid, redundáns teszteket.
- Annotáció alapú: ha a kód tartalmaz feltételeket a bemenetre és kimenetre, és egyéb annotációkat, ezek alapján generálhatóak tesztek.
- Keresés alapú: (Search-based Software Engineering, SBSE) Metaheurisztikákat alkalmazó algoritmusokkal dolgozik, genetikus, szimulált lehűtés, hegymászó, stb. A problémát egy keresési feladatként fogja fel, ahol a keresési tér a program felépítése és lehetséges bemenetei, a célfüggvény pedig elérni a teszt célt (pl teljes döntés lefedettség). Az Evosuite minden teszt célt számításba tud venni, többféle metrika alapján generál teszteket, pl. magas lefedettség minimális esetszámmal. Specialitása hogy minimális tesztkóddal dolgozik, amit olvashatóan generál. Sandboxot használ.

Kihívások: komplex aritmetikai operátorok (pl logaritmus), lebegő pontos számok, nem-triviális string műveletek, környezeti hívások (fájlok, natív kód, külső könyvtár), sokszálú programok, összetett adatstruktúrák, pointer műveletek…

Eszközök: Szimbolikus: KLEE, Pex, Sage, Jalangi, Symbolic PathFinder. Véletlen: Randoop. Annotált: AutoTEst. Keresés: Evosuite.

7 Modell alapú tesztelés: A modell alapú tesztgenerálás alapfeladatai és előnyei. MBT folyamata: modellezés, tesztkiválasztási kritériumok, generálás és végrehajtás. Eszközök.

Feladatok: Általános folyamat: a követelményekből létrehozunk egy absztrakt tesztmodellt, valamint választunk egy teszt célt, hogy milyen és mekkora lefedettséget szeretnénk elérni. A modell és a cél figyelembe vételével állítjuk elő az absztrakt testeseteket, azokból a konkrét teszteket, amiket lefuttatunk a rendszerre, és kiértékeljük az eredményeket.

- Absztrakt teszteseket generálása az absztrakt modellekből
- Konkrét tesztek generálása az absztrakt tesztesetekből
- Tesztek végrehajtása automatikusan vagy manuálisan.

Előnyök:

- Átlátható információk: segíti a kommunikációt az érdekeltekkel, megérteni a fogalmakat és követelményeket.
- Korai tesztelést segíti: specifikáció könnyen ellenőrizhető, szimulációkra használható, teszt adatok generálására is használható.
- Magasabb absztrakciós szinten kezeljük a rendszert, ami könnyebb, mint a komplex rendszerrel való munka.
- Tesztek automatikusan végrehajthatók.

MBT folyamata: Model Based Testing, tesztelés modelleken alapulva vagy modellek segítségével. Nem csak teszteléshez, nem csak automatikus végrehajtáshoz, és nem csak modell-vezérelt fejlesztéshez használható!

- Modellezés: Azt modellezzünk, ami a tesztelés szempontjából fontos: pl. funkcionalitást, teljesítmény faktorokat, stb. Az absztrakciós szint megválasztása körültekintést igényel, a túl sok és a túl kevés részlet sem hasznos. Érdemes a különböző tesztelési szempontokhoz különböző modelleket készíteni. Strukturális és működést leíró modellezési nyelvet is választhatunk. Szükség lehet a rendszer modellezésére, vagy usage modellre, ami a használatát modellezi, a környezettel és a felhasználókkal, a rendszer bemeneteivel. Egy-egy külön tesztesethez is készíthetünk modellt. A fejlesztéshez is szükség lehet modellekre, ezeket azonban nem mindig lehet közvetlenül felhasználni a tesztelés segítésére. Érdemes külön kezelni a tesztelői és fejlesztői modelleket.
- Tesztkiválasztási kritériumok: Tűzhetünk ki célul megfelelő lefedettséget, követelményeknél, modell elemeknél (állapot, átmenet, döntés, feltétel, stb), vagy adatokkal kapcsolatos lefedettséget. Választhatunk teszteseteket sztochasztikusan, véletlenszerűen is. Scenario vagy minta-alapú kiválasztással, vagy egyéb, projekttől függő kritériumok alapján, pl. kockázat, effort, erőforrások tesztelésére.
- Generálás: Teszt generálás különböző módszerekkel. Gráf algoritmusokkal, pl teljes állapotátmenet lefedettség utazó ügynök probléma megoldásával. Véges állapotgépekből generálás, párhuzamosított folyamatok, állapotok ellenőrzésére. LTS tesztelés: (least trimmed squares) majdnem legkisebb négyzetes hiba, de az outliereket próbálja kiszűrni. Modellellenőrzők is használhatóak.
- Végrehajtás: Az absztrakt tesztesetek csak logikai kifejezéseket írnak le, nem konkrét értékeket. Magas absztrakciós szintű eseményeket és akciókat tartlmaznak. A konkrét teszteseteket már konkrét bemeneti értékkel határozzuk meg, és részletes végrehajtási folyamattal, ami manuálisan vagy automatikusan hajtódik végre. Automatizálás esetén egy adaptációs réteg biztosít kapcsolatot a teszt alatt álló rendszerrel, ő közvetíti az eseményeket és akciókat a rendszer felé.

Eszközök:

- *GraphWalker* véges állapotgép készíthető vele, egyszerű őrfeltételekkel, beállítható állapot, átmenet és időlimit (véletlen séta) feltétel. A gráfot képes bejárni véletlenszerűen, A* bejárással, vagy a legrövidebb utat kiválasztva. JUnit tesztek generálhatóak belőle.
- *Conformiq* egy ipari eszköz, állapotgépeket és Java kódot tud kezelni, követelménylefedettséget, állapot és átmenet lefedettséget tud figyelembe venni. Sok egyéb eszközzel integrálható.
- *SpecExplorer* szintén egy ipari eszköz, C# modell program és adapter kód segítségével használható

8 Regressziós tesztelés: Regressziós tesztelés megközelítései. Regressziós tesztek kiválasztása. Tesztek osztályozása. Mohó algoritmus.

Részleges újratesztelése az egész rendszernek vagy egy komponensnek, hogy megállapítsuk, hogy a módosításoknak nem volt következménye az eddigi működésre, és a rendszer vagy komponens még mindig megfelel a specifikált követelményeknek.

Tesztek kiválasztása: Kihívás: mit teszteljünk? Default stratégia: mindent… Túl sok teszt esetén azonban kicsúszhatunk a tesztelésre szánt időkeretből, kis módosítás esetén overkill, és lassan kapunk visszajelzést az eredményről. Tradeoff, kompromisszum elérésére kell törekedni a precízitás és az erőforrások között. Van egy tesztkészletünk (T), és elemek, amiknek a lefedésére törekszünk (C) (követelmények, kód struktúra, stb.). A T->C hozzárendelés segítségével választunk tesztkészletet. A gyakorlatban heurisztikákkal oldják meg.

- Tesztkészlet minimalizálása: redundáns teszteket kihagyjuk. Minimális lefogó ponthalmaz probléma, ami NP-nehéz.
- Tesztesetek kiválasztása: releváns teszteket választjuk ki, esetleg kicsit módosítjuk őket. Minimal set cover probléma, szintén NP-nehéz.
- Tesztesetek priorizálása: újrasorrendezzük a teszteket valamilyen cél szerint, pl gyorsaság, hibadetektálás.

Tesztek osztályozása:

- Újrahasználható tesztek: a rendszer változatlan részeit tesztelik.
- Újratesztelendő tesztek: amik a rendszer módosított részeit tesztelik, vagy amely teszteket módosítottak.
- Elavult tesztek: amiket már nem tudunk használni a specifikáció vagy a rendszer struktúrájának változása miatt.
- Új struktúra tesztek: a lefedettség javítására létrehozott új tesztek, a rendszer új stuktúrájához.
- Új specifikáció tesztek: amik a specifikációban megjelent új követelményeket ellenőrzik.

A regressziós teszt során az újratesztelendő és az új teszteket érdemes futtatni, vagyis a módosított és új kódot ellenőrizzük.
Cél lehet a követelmény lefedettség (nyomonkövethetőség miatt), vagy strukturális lefedettség követelmények (komponens, fájl, osztály, metódus, sor). Itt is megjelenik, hogy arany középutat kell találnunk a precizitás és a gyorsaság között.

Mohó algoritmus:

1. Ha egy követelmény csak egy teszt által van ellenőrizve, azokat mindenképp belevesszük a tesztkészletbe.
2. Amíg vannak lefedetlen követelmények, kiválasztjuk azt a tesztet ami a lehető legtöbbet fed le a lefedetlen követelmények közül.
   Visual Studio pl. tud ilyet.

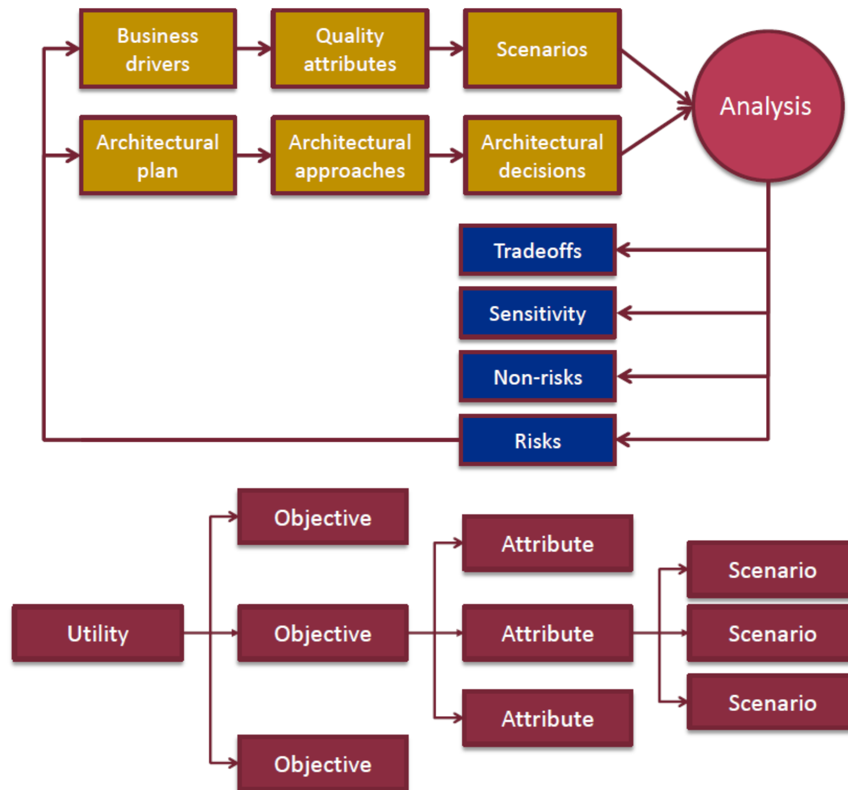
9 Architektúra ellenőrzése: Architektúra leírása, nyelvek. ATAM. Szisztematikus átvizsgálási módszerek (interfész analízis, hibahatás analízis). Modell alapú vizsgálatok (megbízhatóság, teljesítmény analízise).

Architektúra leírása: Az architektúra rendszert alkotó komponensek, tulajdonságaik, kapcsolataik leírása. Architektúra tervezésnél számos tervezői döntést kell hoznunk. Kiválasztani a komponenseket és definiálni a kapcsolatukat. A rendszer funkcionalitása a komponensek (szoftver-hardver) együttműködése által valósul meg. A komponensek tulajdonságai befolyásolják az extra-funkcionális tulajdonságokat (teljesítmény, megbízhatóság, tesztelhetőség, stb).

Nyelvek:

- UML: strukturális és viselkedési diagrammok is
- SysML: mérnöki modellek, pl. block diagram.
- AADL Architecture Analysis and Design Language: beágyazott rendszerekhez. Szoftver részről modellezhetőek a folyamatok, szálak, megosztható adatok, alprogramok. Hardver részről modellezhetőek a processzorok, memória, buszok, külső eszközök. Modellezhető a szoftver és hardver komponensek kapcsolata is. A modell kiértékeléséhez fontos tulajdonságok is megadhatóak, időzítések, ütemezések, stb. A modellek használhatók grafikus, szöveges és XML formátumban is.

ATAM (Architecture Trade-off Analysis Method): azt vizsgáljuk, hogy adott minőségi követelményeknek megfelel-e az architektúra, a tervezői döntések hogyan támogatják a minőséget, és milyen kockázatok vannak. Szisztematikusan összegyűtjük a minőségi célfüggvényeket és tulajdonságokat egy hasznossági fában, priorizálva. Majd kiértékeljük a tervezői döntések gyenge pontjait, kompromisszum lehetőségeket és kockázatokat.



Utility tree felépítése: A hasznosságot minőségi célfüggvények írják le. Ezeket különböző tulajdonságok befolyásolják. A tulajdonságokhoz scenáriókat veszünk fel, ahol leírjuk, adott esetben hogy alakul az adott tulajdonság. (Pl. tulajdonság: hw hiba, scenarió: lemez hiba esetén 5 percen belül induljon újra a rendszer)

Analízis lépései:

1. Ellenőrzés, hogy a fában felvett scenráriókra az architektúra megfelelő működést biztosít e. Pl. hiba kezeléshez tartalmaz e elég tartalék erőforrást.
2. Gyengepontok vizsgálata. Pl. sok tartalék adatbázis növeli a rendelkezésre állást, de szinkron adatfrissítés esetén lassítja a rendszert, aszinkron esetén viszont fennáll az esélye az adatvesztésnek.
3. Kompromisszum lehetőségek vizsgálata, optimalizálás
4. Kompromisszum lehetőségek kockázatainak vizsgálata

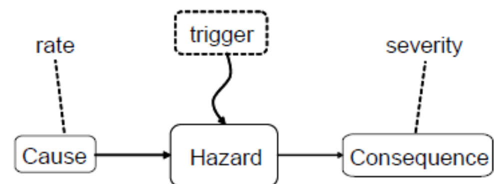
ATAM folyamata:

1. A kiértékelés vezetője ismerteti a módszert
2. Vezető fejlesztő ismerteti az üzleti részleteket: funkciók, minőségi célok, érdekeltek, követelmények.
3. Tervezők ismertetik az architektúrát
4. Tervezők azonosítják a tervezői döntéseket
5. Tervezők és ellenőrzők megalkotják a hasznossági fát. Létrehozzák a szcenáriókat, bemenetekkel, hatásukkal, amik fontosak a minőségi célok szempontjából. Priorizálják a szcenáriókat.
6. Ellenőrzők megvizsgálják az architektúrát a szcenáriók alapján, meghatározzák a gyengepontokat, kompromisszum lehetőségeket és kockázatukat.
7. Érdekeltek kibővítik a szcenáriókat, brainstorming a lehetőségekről.
8. Ellenőrzők folytatják az architektúra értékelését
9. Végeredmény ismertetése az ellenőrzők által, felkészülés a záró összegző dokumentumok készítéséhez.

Előnyök: érthető és átlátható minőségi célok a célfüggvények, szcenáriók és prioritások által. Kockázatok korai azonosítása, kompromisszum lehetőségek körüljárása. Hasznos, hogy az érdekeltek is be vannak vonva a folyamatba, tervezők-tesztelők-felhasználók-ellenőrzők, és kommunikáció zajlik a felek között. Megfelelő dokumentációt biztosít az architektúra tervezés során hozott döntésekről és kockázatukról.

Szisztematikus módszerek:

- Interfész analízis: Célja a komponens interfészek megfelelőségének vizsgálata. Szisztematikusan vizsgálja a kapcsolatok és interfészek lefedettségét. Szintaktikus vizsgálattal a funkciók paramétereinek számát és típusát vizsgálja. Szemantikus vizsgálattal a specifikáció és a komponensek funkciójának leírása alapján ellenőrzi az architektúrát. Viselkedési vizsgálattal a működést leíró specifikáció alapján ellenőrzi a komponenseket.
- Hibahatás analízis: Cél a hibák hatásának és az architektúra működésében veszélyt jelentő események vizsgálata. Mik az okai és mik a következményei a hibáknak? A veszélyeket katalógusban írja le, kategorizálja, következményeit és gyakoriságát is leírja (-> kockázati mátrix). Használható a kockázat csökkentéséhez.



Kategorizálási technikák: szisztematikus módszerekkel.

- Ok-okozati nézet: előre nézve az eseményeket, a következményeket vizsgálhatjuk (inductive), visszafelé pedig az okokat (deductive).
- Rendszer hiearachikus nézete: bottom-up módon, komponensektől kezdve, alrendszereken át a teljes rendszerig, vagy top-down módon a legfeleső szintről felbontva komponensekre a rendszert.

Hibafa: rendszer szintű hibák okainak vizsgálata, top-down módon felbontva a lehetséges okokat. Segít azonosítani a komponens szintű hibák és események kombinációját, amik összességében rendszer szintű hibát okoznak. A hibát okozó eseményeket AND és OR kapukkal addig bontjuk szét apróbb okokra, amíg elemi hibaokokat kapunk, ők lesznek a fa levelei, az elsődleges események, amik rendszerhibát okozhatnak. A fa gyökere a rendszer szintű hiba (pl. beragad a lift, szétbontható OR kapuval beragadt a gomb/áramszünet/rendszerhiba okokra, azokat még tovább, stb.) Fa redukálható, aminek a végeredménye, hogy megkapjuk a SPOF (single point of failure) hibaokokat. Hibaeseményekhez hibalehetőséget adva meghatározható a meghibásodás valószínűsége a rendszernek.

Eseményfa: azt veszi sorra, hogy az események milyen kombinációja okoz hibát. Kezdeti esemény, komponens szintű, majd megy végig a kapcsolódó eseményeken, más komponensek eseményein/hibáin. Aktualitás/időzítés alapján veszi őket sorba. Az alapján bontjuk ágakra a fát, az adott szintet jelző komponens esemény fennáll-e vagy sem. Leolvasható belőle az utak valószínűsége. A veszélyes esemény sorozatok jól vizsgálhatóak általa. A komplexitás és az események számossága korlátozza a felhasználhatóságát.

Ok-okozati analízisnél együtt kezeljük az esemény- és hibafákat. Az események sorozata az események okával együtt látható, együtt a forward-backward analízis lehetőség. Hátránya a komplexitás, különböző diagram szükséges minden kezdeti eseményhez!

FMEA (Failure Modes and Effects Analysis) komponensek táblázatos vizsgálata, hiba módok, lehetőségek és hatások. Előnye az átláthatóság, szisztematikus rendszer a hibák összeírásának, hátránya hogy a hibák hatását nehéz átlátni.

Modell alapú vizsgálatok: architektúra kiértékelése a választott megoldások alapján. Analízis modelleket hozunk létre az architektúra modell alapján, pl. teljesítmény modell, megbízhatósági modell, stb. Ezek matematikai modellek, amik kifejezik, a lokális paraméterek hogyan befolyásolják a rendszertulajdonságot. A választott modell karakterisztikájából származtatható a rendszertulajdonság. Az analízis modellek létrehozása az architektúra modellből (komponensek és kapcsolataik), a komponensek paramétereiből, a komponensek kapcsolatainak paramétereiből, és a komponensekhez és kapcsolataikhoz tartozó analízis modellekből valósul meg.

- Megbízhatóság: Komponens paraméterek például a hibaráta, hibák között eltelt idő, helyreállításig szükséges idő, komponensek kapcsolatának paraméterei a hiba terjesztésének valószínűsége vagy a helyreállítási stratégia, modellezhető Markov lánccal vagy Petri hálóval, vizsgálható rendszertulajdonság a megbízhatóság, rendelkezésre állás, MTTF, MTTR, MTBF
- Teljesítmény: komponens paraméterek például a funckiók végrehajtási ideje, prioritásuk és ütemezésük, komponensek kapcsolatának paraméterei a hívás továbbadási ráta és a szinkronitás, modellezhető queuing network-kel, vizsgálható rendszertulajdonságok a válaszidő, áteresztő képesség, processzor kihasználtság.

10 Megbízhatósági analízis: Szolgáltatásbiztonság jellemzői és metrikái. Megbízhatósági blokkdiagramok felépítése és használata. Markov láncok használata a megbízhatósági analízisben.

Dependability - Megbízhatóság: a képesség hogy egy szolgáltatást biztosítsunk amiben indokoltan bízhatunk, hogy teljesíti az igényeinket.

Szolgáltatásbiztonság jellemzői:

- Availability – rendelkezésre állás: a valószínűsége hogy a rendszer megfelelően működik
- Reliability – megbízhatóság: a valószínűsége az egy huzamban folyamatosan jól működő rendszernek
- Safety – biztonság: mentes a váratlan hibák kockázatától
- Integrity – integritás: hibát okozó változtatásoktól mentes
- Maintainability – karbantarthatóság: a javítások és fejlesztések megvalósíthatóak.

Szolgáltatásbiztonság metrikái:

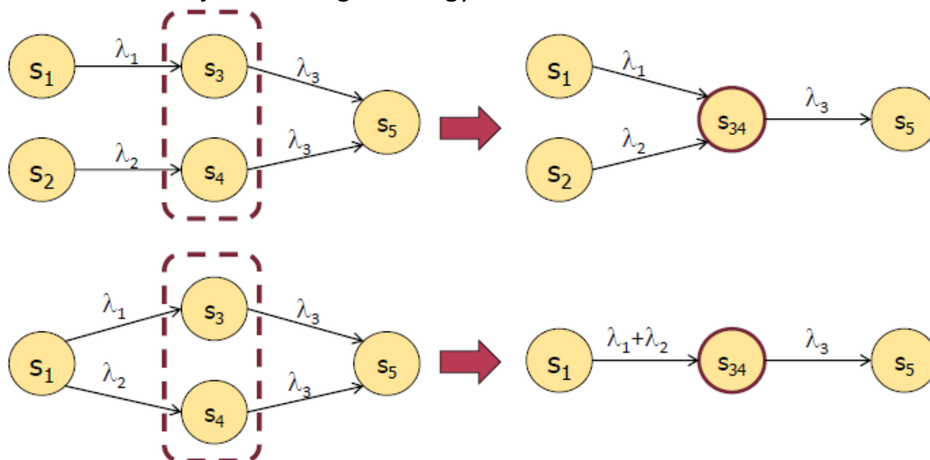
- MTFF – Mean Time to First Failure
- MUT – Mean Up Time (Mean Time To Failure)
- MDT – Mean Down Time (Mean Time To Repair)
- MTBF – Mean Time Between Failures
- Availability – a(t) = P{ s(t) ϵ U } (U = Up)
- Reliability: r(t) = P{ s(t') ϵ U, ∀t' < t}

Megbízhatósági analízis célja a komponens karakterisztikák vizsgálata (hibaráta, meghibásodás valószínűsége, megbízhatósági függvény), és belőlük a rendszer karakterisztikájának kiszámítása. Az analízis eredménye aztán felhasználható a különböző alternatívák összehasonlításához, érzékenységi vizsgálathoz (mi a hatása annak, ha egy másik komponenst választunk, melyik komponenseket cseréljük ki a jobb eredmény elérése érdekében, stb), megbízhatósági tulajdonságok megállapításához (pl. tanúsítvány szerzéséhez).

Megbízhatósági blokkdiagram: Komponensek két állapota létezik: *hibás* vagy *hibamentes*. A komponensek állapota független a többi komponenstől. Komponensek közötti kapcsolatot ábrázolja: *milyen típusú redundancia* van a rendszerben? *Soros* vagy *párhuzamos*, soros esetén a komponensek nem redundánsak, az összes komponens szükséges a rendszer működőképességéhez. Párhuzamos kapcsolat esetén a komponensek helyettesíteni tudják egymást hiba esetén. A diagramban a blokkok a komponensek, az összeköttetések a komponensek közötti kapcsolat (párhuzamos vagy soros), az útvonalak egy-egy rendszer konfigurációt adnak ki. A rendszer működőképes, ha van egy olyan útvonal a kezdőponttól a végpontig, amin minden komponens hibamentes.
Soros: $r_R(t) = \prod_{i=1}^{N} r_i(t)$, MTTF = $1 / \prod_{i=1}^{N} \lambda_i$
Párhuzamos: $1 - r_R(t) = \prod_{i=1}^{N} 1 - r_i(t)$, MTTF = $(1 / \lambda) \sum_{i=1}^{N} 1/i$

Markov láncok: **CTMC = (S, R)**, ahol **S** diszkrét állapotok egy halmaza, **R** pedig a tranzíciós ráták, annak az értéke, hogy 1 időegység alatt mekkora valószínűséggel tüzel az állapotátmenet. (R: SxS -> valós nemnegatív szám). Egy mátrixban leírható. Az **infinitesimal generátor mátrix** a **Q = R – diag(E)**. **σ**-val jelölünk egy **$s_0,t_0,s_1,t_1,\ldots$ útvonalat**, állapotok és állapotátmenteke egymás utániságát. **σ@t** jelöli, hogy melyik állapotban voltunk a **t** időpillanatban.

A CTMT kiértékelésével a célunk megállapítani, hogy egy s_0 kezdőállapotból indulva t idő múlva melyik állapotban mekkora valószínűséggel van a rendszer (**tranziens állapot valószínűség**, **$\pi(s_0, s, t)$**)! A **steady state valószínűségeket** keressük, ami egy egyenletrendszer megoldásával kapható meg.
Megbízhatósági analízisnél a Markov lánc állapotai a rendszer állapotai: a komponensek állapotainak kombinációi (hibamentes, vagy hibás valamely módon). A tranzíciók rátái a **komponensek hiba- és megjavulási rátái**.
A rendszer szintű tulajdonságok megállapításához először meghatározzuk azokat az állapotokat, melyek a rendszer **Up** állapotai. A Markov lánc kiértékelése után adottak a tranziens állapot valószínűségek, és a steady state valószínűségek. A **rendelkezésre állás** ekkor a tranziens állapot valószínűségek összege minden Up állapotra, az **aszimptotikus rendelkezésre állás** ugyanez a steady state valószínűségekkel, a **megbízhatóság** ugyanaz mint a rendelkezésre állás, de előtte módosítani kell a modellt, minden Down állapotból Up állapotba menő élt törölni kell!
Állapotok egyesítésével **redukálható** a Markov lánc, ha két állapotból ugyanabba az állapotba megy él ugyanakkora rátával. Ha ugyanabból az állapotból ment beléjük él, az új, egyesített állapotba a két él rátájának összegével megy él.



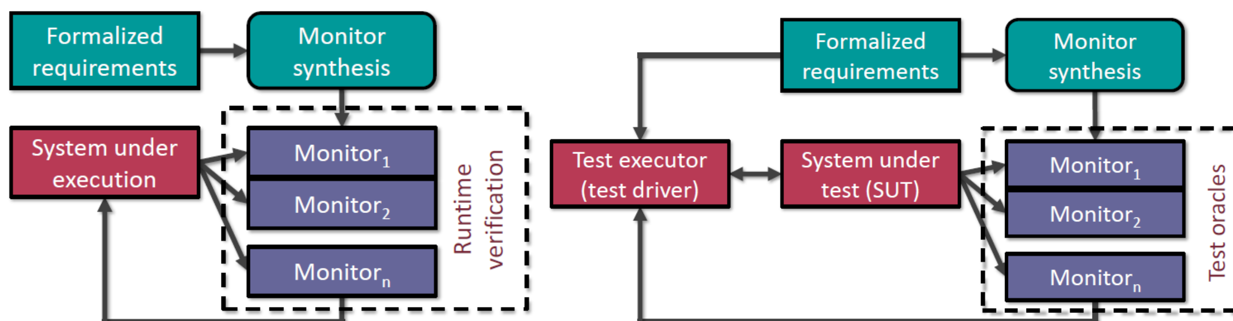
Hot redundancia esetén MTTF = 1/(kλ), ahol k komponens jó
Bővíthető a rendszer **rewardokkal** is, ez valamilyen profitot vagy költséget fejez ki, amik hozzárendelhetők tranzakciókhoz vagy állapotokhoz. Például időegységenként 300 profitot jelent, ha a rendszer hibamentes.
Analízisnél a **rewardok összegére** lehetünk kíváncsiak, vagy egy **reward pillanatnyi értékére** (pl. egy óra múlva mennyi a rendszer működtetési költsége), vagy a **steady-state reward értékekre**, vagyis hogy a végtelenbe nézve átlagosan mennyi lesz a költség/profit.

11 Futásidőbeli verifikáció: Célkitűzések és használati esetek. Futásidejű verifikáció referencia automaták, temporális követelmények vagy szekvencia diagramok alapján.

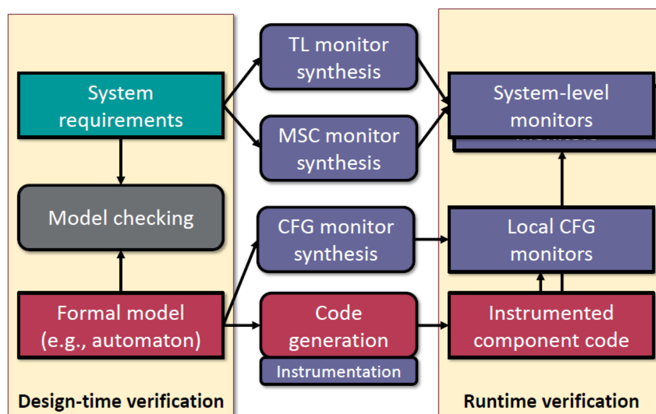
Futásidőbeli verifikáció: futási időben ellenőrizni a rendszer működését, formálisan specifikált tulajdonságok alapján. Biztonság-kritikus rendszereknél ellenőrizhető, hogy nem állnak fenn veszélyes helyzetek, vagy tanúsítvány biztosításához ellenőrizhető a megfelelő rendelkezésre állási szint (SLA). Ezen kívül vannak elkerülhetetlen futásidejű hibák, mint a véletlen hardver hibák, vagy szoftver konfigurációs hibák, ezért a futásidejű ellenőrzés nagyon fontos.

Célok: A futásidejű hibadetektálás az alapja a hibakezelésnek. A forráskód alapján operációs hibákra derülhet fény például vezérlési folyam gráfok vizsgálatával, a követelmények teljesülését vizsgálva szisztematikus módszerekkel ellenőrizhetjük a rendszert, architektúra tervet, formalizált követelményekkel leírva a követelményeket pedig automatikusan kiértékelhető futási időben a rendszer működése, amit a monitorozó alkalmazások tesznek lehetővé.

Használati esetek: a monitorok használhatóak *futásidejű verifikációra*, formalizált követelményeket értékelnek ki, és operációs hibákat, konfigurációs hibákat, váratlan környezeti feltételeket detektálnak.
Ezen kívül a monitorok lehetnek *teszt orákulumok* is a tesztelő keretrendszerekben, a megadott követelmények teljesülésének ellenőrzését vizsgálhatják, és tervezési vagy implementációs hibákat detektálhatnak.



Kihívások: Verifikációs technikák: végrehajtás során a temporáris kifejezések ellenőrizhetők *temporális logikával*, *referencia automatával*, *reguláris kifejezésekként*. Végrehajtható assert-ek is lehetnek a kódokban. Specifikáció nélküli monitorozása is lehetséges az általános hibáknak, deadlock, livelock, sorosítási konfliktusok, stb.
Aktív és *passzív műszerezéssel* is megoldható a futásidejű verifikáció. Aktív ha kódrészletekkel bővítjük ki a forráskódot, passzív ha módosítás nélkül figyeljük meg a rendszert. Aktív módszer az AOP (Aspect-Oriented Programming). Szinkron és aszinkron módon is monitorozhatunk.



Referencia automaták: Tényleges viselkedés (programfutás) összevetése referencia viselkedéssel (modell vagy specifikáció). Adott az ellenőrzés alatt álló komponens. Belső működése környezetfüggetlen nyelvtannal megadva (CFG, Context Free Grammar). A komponensben operációs és implementációs hibákat tudunk detektálni. A komponens futásidejű verifikációjához olyan felműszerezése szükséges, ami segítségével a monitorozó komponens informálódik a komponens állapotáról. A monitorozó komponens a referencia automata alapján detektálja a hibákat, CFG segítségével. Lényegében ellenőrzi, helyes állapotban van-e a komponens a referencia automatához képest, beragadt-e valahol, betartja-e az időzítési feltételeket.

Temporális követelmények alapján: Temporális logika boolean operátorokkal (és vagy negálás implikáció), temporális operátorok (X, F, G, U), útvonal operátorok (E, A).
Állapotok és események sorrendjét, elérhetőségét vizsgálhatjuk. Biztonsági tulajdonságok: *invariánsok teljesülése* minden állapotban. Élőségi tulajdonságok: szükséges *állapotok elérhetősége*. Futási időben ellenőrizhető lineáris idejű temporális logikával az adott lefutása a programnak, elágazó idejű temporális logikával pedig a tesztelés során bejárt útvonalak (cél lehet az összes út lefedése a tesztelés során). A felműszerezett alkalmazások futási időben adnak információt a CTL monitornak, aminek kódja a CTL követelményekből *monitor szintézissel* generálható: követelmények alapján megfigyelő automatát konstruálunk a futásidejű szekvenciákhoz, bemenetek a jelzőszámok a hivatkozott lokális feltételek teljesüléséről, szekvencia kezdete és vége, kimenet pedig hogy elfogadott, hibás vagy nem meghatározott a szekvencia eredménye. Teszt orákulumként használható.
LTL követelményekhez monitor szintézis: alap gondolat, hogy két elfogadó automatát csinálunk, ami elfogadja a φ kifejezéseket, és egyet ami elfogadja φ negáltjait! *Elfogadó állapot*, ami elfogadja az eredeti követelményt, *elfogadható állapot*, ahonnan van olyan eseménysorozat, ami elfogadó állapotba visz. Monitorozás eredménye false, ha nem elfogadható állapotban ért véget a lefutás, true, ha a negált kifejezésekhez létrehozott automata nem elfogadható állapotban állt meg (tehát nem érhető el olyan állapot belőle, ahol a követelmények biztosan nem teljesülnek), és nem meggyőző, ha mindkét automatában elfogadható állapotban vagyunk. Monitor szintézis folyamata: a két automata egy szorzatát képezni, véges állaptgépet készíteni, majd minimálisra redukálni.

Szekvencia diagramok alapján: (MSC, Meassega Sequence Charts) Cél intuitív leírás alapján ellenőrizni a komponensek közötti interakciókat. Szcenárió alapú követelmények esetén hasznos. Szekvencia diagram: életvonal, üzenet, őrfeltételek, különböző fragmentek (alt, opt, par,…) . A szekvencia diagram két részre osztható, ami a *körülményeket* írja le (condition), és ami az *ellenőrizendő interakciókat* (assert). Az MSC monitor szintézis az érdekelt életvonal mentén történik, az interakciókból egy *állapotgépet* konstruálva. A condition és az assert részben mást jelent, hogy nem teljesül egy feltétel. Condition részben csak annyit, hogy nem áll fenn az az élethelyzet, amiben ellenőrizhetővé válna a tesztelt működés, assert részen viszont a rendszer hibás működését jelenti. Ha elérjük az állapotgép végállapotát, azt jelenti, teljesül a feltétel.
Az állapotgépből generálható a monitorozó kód.
Egy ütemezővel használhatjuk a monitorozó komponenst, ami felelős a monitorozás indításáért és leállításáért, és menedzseli a hibajelzéseket és állapotokat. Monitorozás módjai *initial*, *invariant* és *iterative* lehet, csak az elején figyelünk, vagy össze-vissza, vagy meghatározott időközönként.