Klasszikus verifikáció és validáció

Előadásvázlat a „Szoftver verifikáció és validáció” tárgyhoz

Majzik István
Budapesti Műszaki Egyetem
Méréstechnika és Információs Rendszerek Tanszék

2006. március 31.

Tartalomjegyzék

1. Bevezető

A klasszikus verifikációt és validációt alapvetően a biztonságkritikus rendszerekben használt szabványok (pl. IEC 61508, MSZ EN 50128) ajánlásai alapján tekintjük át. Így élvezhetjük az iparszerű alkalmazás és az egységes tárgyalás előnyeit, ugyanakkor láthatjuk a szabványosításkor ”befagyott” lehetőségek korlátait is.
A szabványok tipikusan négy lényeges kérdést rögzítenek:

- Biztonságintegritási szintek: Ez alapján határozható meg a szervezeti rend és a szükséges módszerek.
- Szervezeti rend: Felelősségek és hatáskörök.
- Életciklus és dokumentáció: Az életciklusra nincs szigorú előírás, csak ajánlat.
- Módszerek kiválasztása az egyes életciklus fázisokhoz, a biztonságintegritási szinttől függően.

Ezeket a döntéseket egy *szoftver-minőségbiztosítási tervnek* kell rögzítenie. Ezt a fejlesztés megkezdése előtt el kell készíteni.

2. Szoftver minőségbiztosítás (átfogó)

Cél: Az összes technikai és irányítási tevékenység meghatározása, figyelése és ellenőrzése. Biztosítani kell az előírt védelmet a szisztematikus hibák ellen, követni kell a verifikáció és validáció végrehajtását.
Kimenet:

- Szoftver minőségbiztosítási terv.
  Meghatározza a következőket:
  - *Életciklus modell* (ki/bementek, tevékenységek, továbblépés feltétele) és a *dokumentálási rend.*
  - *Korábbi fejlesztések és ellenőrzések* beillesztésének módszere.
  - *Beszállítók* ellenőrzésére szolgáló eljárások.
  - Teljesítendő *számszerű* minőségi jellemzők.
  - Saját *korszerűsítésének* szabályai is (gyakoriság, felelősség, módszer).
- Szoftver konfigurációmenedzselési terv. Végrehajtása általában a *konfiguráció kezelő testület* feladata (ez a projekt szakmai vezetésből, a vezető verifikátorból és validátorból áll).
  Meghatározza a következőket:
  - Dokumentumok konfigurációs ellenőrzése kibocsátás előtt.
    Jellemző életciklus dokumentumok (pl. munkautasítások) esetén:
    - Megbízás elkészítésre, szerkesztés, elkészítés.
    - Jóváhagyás, életbeléptetés, értesítés.
    - Archiválás, sokszorosítás.
    - Érvénytelenítés, értesítés, törlés.

  - Forráskód konfigurációs ellenőrzése kibocsátás és tesztelés előtt.
    Jellemző tevékenységek a szoftver konfigurációkezelés esetén:
    * Konfiguráció azonosítása (források, dokumentációk, eszközök, hardverek).
    * Módosítás (kivétel majd visszaillesztés a konfigurációba, konfliktusok elkerülése illetve felismerése párhuzamos módosítások esetén).
    * Kiadás (lezárás).
  - Az életciklus környezet (telepítő fájlok, fordítók, debuggerek stb.) rögzítése és kezelése.
  - Illetéktelen változtatások elkerülése (jogosultságok kezelése, tárolási és átviteli hibák kiküszöbölése).
  - Problémabejelentés és javítás eljárásai (ld. karbantartás).

Követelmények:

- Legalább EN ISO 9000 sorozatnak megfelelő minőségbiztosítási rendszer, EN ISO 9001 szerinti akkreditáció ajánlott.

Példa a konfigurációkezelés támogatásához:

- CVS, Concurrent Version Systems: A módosításokat és az azokból következő esetleges konfliktusokat központilag nyilvántartja, és ha kell, értesíti róla a fejlesztőket. A projekt elsődleges példánya (master copy) általában központi szerveren (repository) tárolódik. A fejlesztők innen kérhetnek munkamásolatot, valamint - a módosítások elvégzése után - ide küldik vissza a forráskódot. A verziókezelő legfontosabb műveletei a következők:

  - Login, logout: Jogosultságvizsgálathoz szükséges.
  - Import: Új projekt létrehozásához importálni kell a forrást a verziókezelőbe.
  - Check out: Aktuális másolat kérése a verziókezelőtől.
  - Commit: A módosítás utáni forrás visszaküldése a verziókezelőnek. A forrás a változtatás jellegét tartalmazó loggal van ellátva. Ha egy fejlesztő egy olyan állományt módosított, amit a többi fejlesztő nem módosít, akkor a módosítást elfogadja a rendszer. Ellenkező esetben (pl. felismeri, hogy A és B fejlesztők párhuzamosan módosítanak egy kódállományt és A éppen Commit műveletet szeretne végrehajtani) elutasítja a Commit műveletet, aminek hatására a konfliktust az A fejlesztőnek fel kell oldania a B által végzett módosítások frissítésével és saját kódjába történő befésülésével.
  - Diff: A saját másolat és a szerveren lévő elsődleges változat összehasonlítása és a különbségek megjelenítése.
  - Update: A mások által végzett módosítások saját másolatunkon történő elvégzése. Érdemes egy Commit előtt végrehajtani.
  - Status: Információt kaphatunk arról is, ha az elsődleges verzió és a saját másolat között különbség van (azaz valaki már benyújtotta (Commit) módosításait, csak még nem frissítettük a másolatot).
  - Log: Megadja, hogy ki milyen módosításokat hajtott végre és mikor. A log használja a Commit esetén már fentebb említett, a fejlesztők által a módosításhoz fűzött megjegyzéseket.
  - Add: A parancs segítségével hozzáadhatók újabb (szükséges) állományok a másolathoz, majd utána a Commit segítségével ezek az elsődleges forráshoz adhatók. A fájlok csak a Commit utasítás után kerülnek be a repository-ba (a szerverre).

  - Remove: Állományok törölhetők a másolatból, majd a Commit segítségével a repository-ból is.
  - Elágazás: Egy közös ős elsődleges forrásból több fejlesztési útvonal is indítható. Az egyes ágakhoz és verziókhoz címkék (cédulák) rendelhetők, ezek alapján lehet őket azonosítani és a tartalmukhoz hozzáférni. Ilyen címke a másolathoz vagy (másolat kérése nélkül) az elsődleges forráshoz is hozzárendelhető. A címkékről a Status művelet is tájékoztat.

3. Biztonságintegritási szintek

A verifikáció és validáció módszereit a *szoftvertől megkövetelt* biztonságintegritási szint határozza meg. Ez pedig a *rendszer* biztonságintegritási szint alapján határozható meg (1. ábra).
A rendszerfejlesztés dokumentumai, amik alapján a szoftverfejlesztés indul, a következők:

- Rendszer követelmény-specifikáció
- Rendszer biztonsági követelmény-specifikáció
- Rendszer biztonsági terv
- Rendszer architektúra leírás

Általános szabály: A szoftver biztonságintegritási szintnek legalább akkorának kell lennie, mint a rendszeré. (Kivétel: Ha bizonyítható, hogy a szoftvermodul meghibásodása nem okozhatja a rendszer nem biztonságos állapotba kerülését.) A különböző biztonságintegritási szintű szoftver modulokat majd a szoftver architektúra rögzíti.

4. Szervezeti rend (résztvevők)

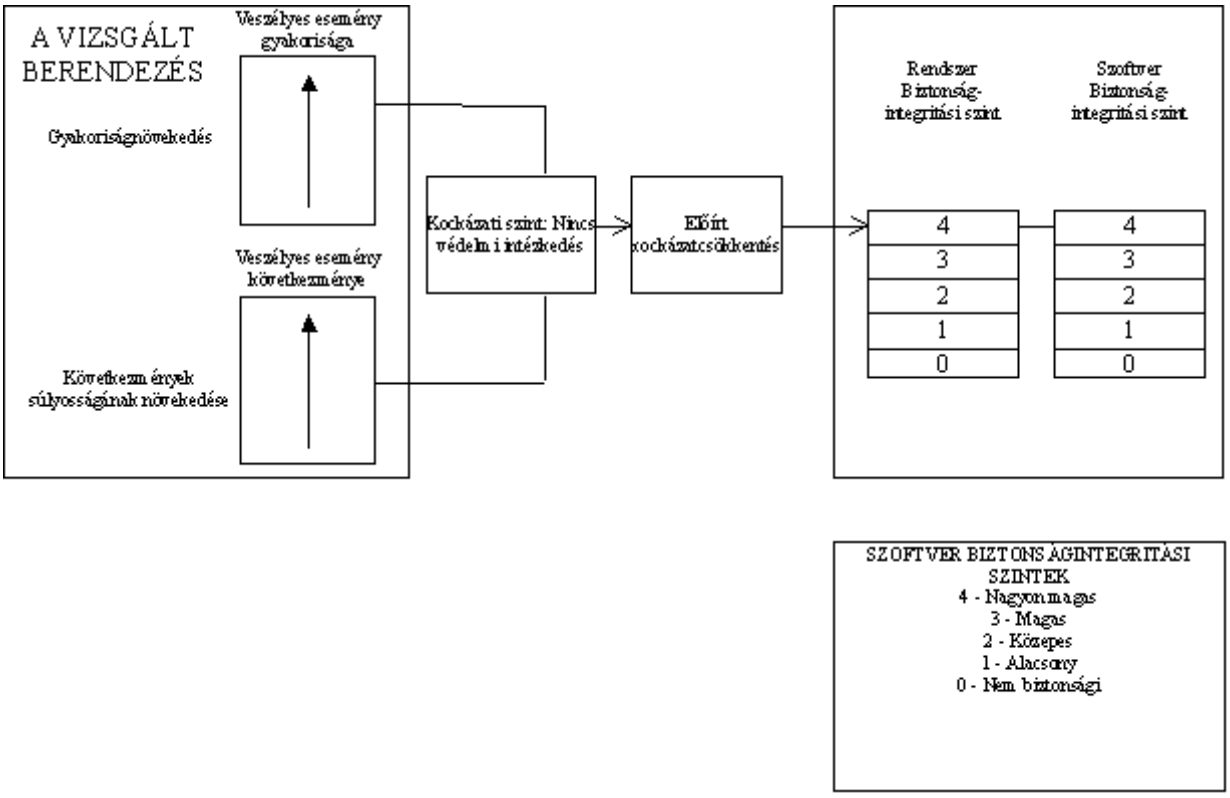
A szereplők a következők (*biztonsági szervezet* irányítása mellett, a képzettségek igazolásával):

- Projekt vezető: A fejlesztés felügyelete.
- Tervező: Specifikáció, architektúra és modultervek, modul implementáció, szoftver integráció elkészítése.
- Verifikátor (igazoló): Modul és rendszer szintű verifikációs feladatok.
- Validátor (érvényesítő): Követelmény-tesztspecifikáció és rendszer validáció elkészítése; a termék átadását letilthatja.
- Értékelő: Fejlesztési folyamat megfelelőségének értékelése; felhatalmazással kell rendelkeznie.

A szereplők függetlenségét a(z) 3. ábra alapján követhetjük:

5. Életciklus és dokumentáció

Az életciklus modell kiválasztása és a hozzá kapcsolódó döntések rögzítése a *szoftver-minőségbiztosítási terv* fő feladata.
Az életciklus modell minden fázisához meghatározott dokumentáció tartozik.



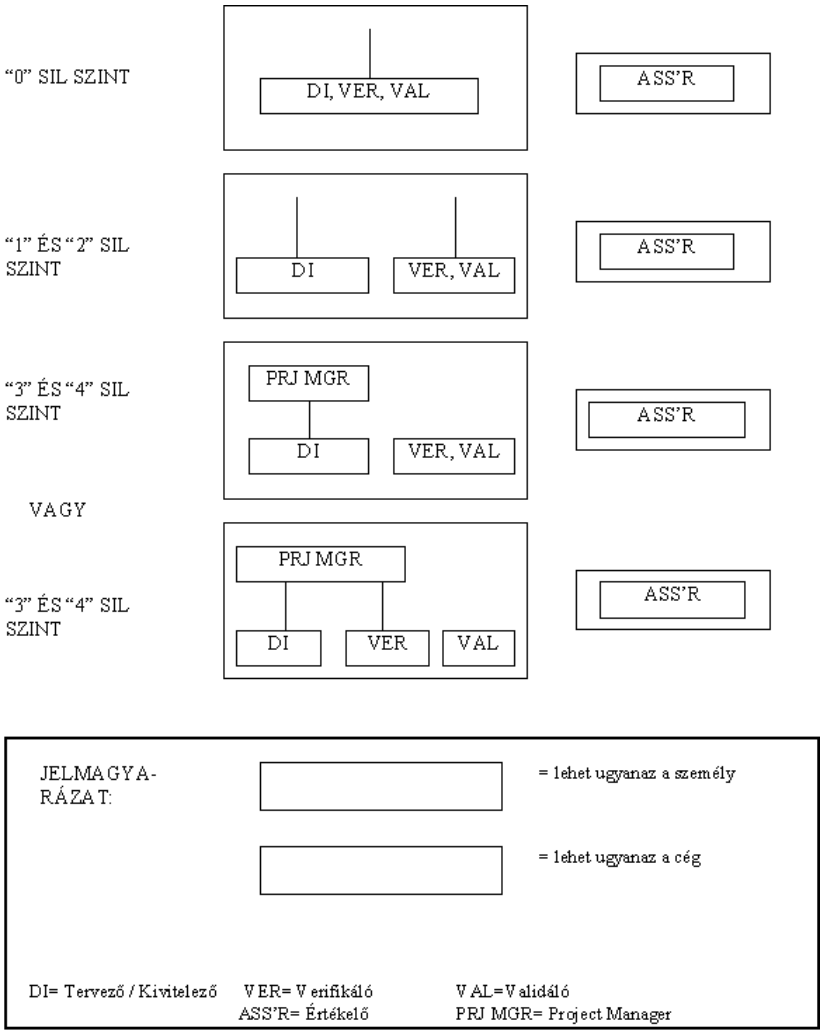
1. ábra. Biztonságintegritási szintek

- A dokumentumok a megelőző fázisok dokumentumaira épülnek (fogalmak, rövidítések, követelmények). Ezeket a kapcsolatokat egy *keresztreferencia táblázat* rögzíti.
- A dokumentumoknak azonosíthatóaknak és feldolgozhatóaknak kell lenniük. Megvalósítás: Dokumentumok is a verziókövető (konfiguráció-menedzsment) rendszerbe kerülnek (lásd részletesen ott). A verziókövető rendszer általános elemei:
  - Elágazásokat is tartalmazó verzió-vonalak.
  - Betétel, kivétel, új verzió létrehozása.
  - Konkurens hozzáférés konfliktusainak felfedése.
- A(z) 3. ábra a kötelezően előírt dokumentumokat mutatja az MSZ EN 50128 esetén.

5.1. Generikus szoftver fejlesztése

A *generikus szoftver* olyan szoftver, ami specifikus adatokkal történő paraméterezés után sok helyen felhasználható. A paraméterezés eredményeképpen áll elő a *specifikus szoftver* (termék).
Generikus szoftver fejlesztése esetén az életciklus fázisok feladatai bővülnek a következőkkel:

- Rendszerfejlesztési fázis: Paraméterezés általános tervezése.
- Követelmény-specifikáció: Paraméterezhető funkciók meghatározása.



2. ábra. A szereplők függetlensége

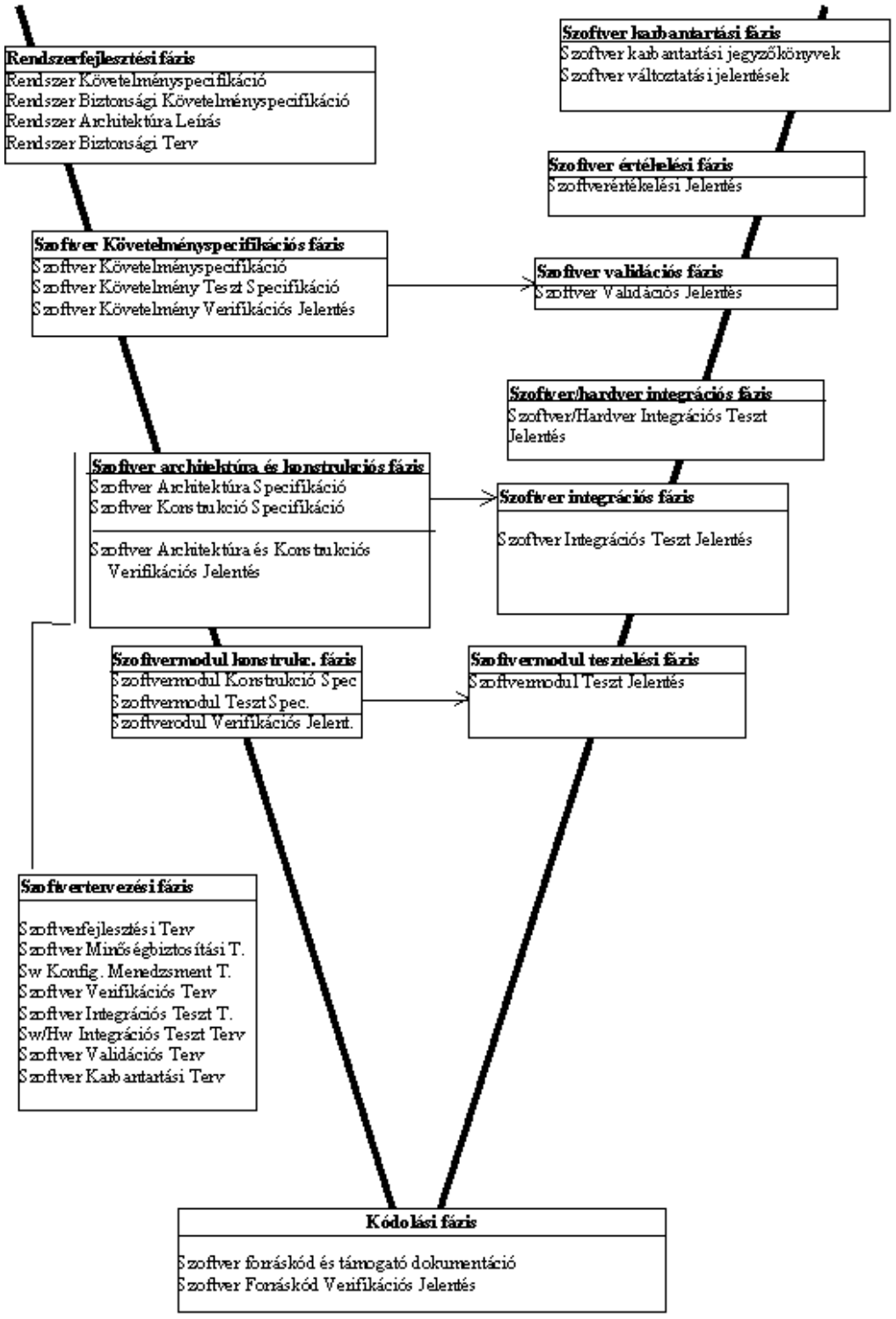
- Architektúra-tervezés és konstrukció: Interfészek tervezése a generikus funkciók és a paraméterek között (pl. konfigurációs állományok, képek stb.); érvénytelen paraméter-kombinációk felismerésének biztosítása.
- Modul konstrukciós fázis: Programkód és paraméter adatok közötti merev elkülönítés biztosítása.
- Verifikációs és validációs fázisok: Minden paraméter-kombináció vizsgálatának biztosítása.
- Karbantartás: A programkód illetve a paraméterek változtatásai közötti összeférhetőség biztosítása.

A specifikus termék esetén a szoftver integráció fázisban történik a konkrét paraméterek integrálása, amit az integrációs tesztek, szoftver-hardver integráció, validáció és értékelés követ. A specifikus termék esetén a következő dokumentáció készül el:

- *Alkalmazási követelmények specifikációja*: Hol használható a generikus termék.
- *Adat-előkészítési terv*: Paraméter adatok összegyűjtése, kidolgozása.

- *Adatteszt terv*: Az adatelőkészítés és paraméterezés ellenőrzésének tervezése.
- *Adatteszt jelentés*: Az adattesztek (paraméterezés megfelelőségének vizsgálata) eredményei.

A dokumentáció készítése szempontjából szokásos megoldás a generikus termék esetén dokumentum sablonokat készíteni, amik mintegy ”kitölthetők” az egyes konkrét, felparaméterezett specifikus szoftverek esetén.



3. ábra. A fejlesztési ciklus dokumentációi

6. Követelmény-specifikáció

Célok:

- Teljes követelménysor meghatározása; a szoftvermérnökök részére szükségtelenné teszi, hogy a követelményeket más dokumentumokból gyűjtsék össze.
- Az alkalmazandó megoldásokat (architektúra, algoritmus stb.) nem tartalmazza.
- A funkcionalitás mellett a nem-funkcionális (teljesítmény, megbízhatóság, biztonság, hatékonyság, felhasználhatóság, karbantarthatóság) rögzítése is feladat.

Bemenet:

- Rendszerszintű (követelmény)-specifikáció, biztonsági (követelmény)-specifikáció, architektúra leírás.
- Szoftver-minőségbiztosítási terv.

Kimenet:

- Szoftver(követelmény)-specifikáció
- Szoftver(követelmény)-*teszt*specifikáció
- Szoftver követelmény verifikációs jelentés.

Jellemzők:

- Bármely későbbi életciklus-fázisban megérthető leírás.
- Külső kapcsolatok, illeszkedések, környezeti feltételek rögzítése.
- A tesztspecifikáció rögzíti minden követelmény ellenőrzését (a rendszerverifikáció során). Tipikusan tesztesetek megadása történik: bemenet, kimenet, teljesítési feltételek.
- Követelmények követésének alapját képezi.

Alkalmazható módszerek (a természetes nyelvi leírás *mellett* ezek közül kell egyet választani):

- Formális módszerek (CCS, CSP, HOL, LOTOS, OBJ, temporális logika, VDM, Z, B)
- Félformális módszerek
- Strukturált metodika (JSD, MASCOT, SADT, SDL, SSADM, Yourdon)

6.1. Formális módszerek

A matematikai alapokon nyugvó formális specifikáció szerepe többszörös. Egyrészt elkészítése hozzájárul a fejlesztett rendszer alapos, precíz elemzéséhez, a működés végiggondolásához, a hibák és ellentmondások korai felismeréséhez. Sok esetben a formális specifikáció a megrendelő és a tervező közös nyelve lehet (különösen, ha a specifikációs nyelvnek van jól áttekinthető grafikus reprezentációja). Ugyanakkor a formális specifikáció támogatja a tervezés későbbi fázisaiban az automatikus elemzést és feldolgozást, pl. a verifikáció és a kódgenerálás szempontjából. A használt formalizmusok (nyelvek) csoportosítása a következő lehet:

Modell-orientált nyelvek: Ezek a nyelvek a rendszer matematikai modelljének konstruálásával adják meg a specifikációt. Ez az állapotok (meghatározott halmazelméleti struktúrák) és az állapoton elvégezhető, új állapotot eredményező műveletek definiálásával (elő- és utófeltételek megadásával) történik. Tipikusan a programozási nyelvhez hasonló adatstruktúrák illetve függvények leírását támogatják, így finomításuk illetve ”fordításuk” az implementáció nyelve felé jelentősen egyszerűsödik (a rendszerállapot leképzése, műveletek finomítása tételbizonyítással ellenőrizhető).
A legismertebb modell-orientált specifikációs nyelvek a VDM, és a Z. A B egy módszer (a Z kiterjesztéseként) a lépésről lépésre történő finomításhoz.

Algebrai nyelvek: Az algebrai specifikációs nyelvek az absztrakt algebra és a kategóriaelmélet módszereit használják a rendszer leírására. Absztrakt adattípusokat (ADT) definiálnak a hordozó halmaz és az értelmezett műveletek megadásával. A műveletek (függvények) megadása elsőrendű logikával történik. E nyelvek előnye az implementációfüggetlenség. Az ADT olyan, mint egy Ada csomag, ahol az operátorok jellemzői láthatóak, de az implementációs részletek elrejtettek.
Ismert algebrai nyelv az OBJ.

Processz leíró nyelvek: A processz leíró nyelvek elsősorban konkurens rendszerek specifikálására alkalmasak. A nyelv kifejezései processzeket írnak le, amelyek operátorokkal egymáshoz kapcsolhatók, egyszerűbb processzekből összeállíthatók (processz algebra).
A legismertebb ilyen nyelvek a CSP és a CCS.

Logikai nyelvek: A logikai specifikációs nyelvek a formális matematikai logikához állnak közel. Gyakran axiomatikus megközelítésben írják le a rendszert. Általában a klasszikus elsőrendű logikára építenek. Jellemző példa lehet a HOL rendszer.
A temporális logikai nyelvek az elsőrendű logikát temporális operátorokkal (pl. mindig, végül is, ennélfogva) egészítik ki, amelyek az állapot-sorrendeken értelmezhetők. Számszerű időkorlátok és időszakaszok megadását általában nem támogatják. Jellemző példák a PLTL, CTL CTL*.

Konstruktív nyelvek: A konstruktív nyelvek a matematika egy ágára, a konstruktív logikai rendszerekre építenek. A megközelítés lényege, hogy a hagyományos matematika függvény fogalma helyett (ami olyan függvényeket is tartalmazhat, amelyek számítógéppel nem értékelhetők ki) csak a hatékonyan számítható függvényekre koncentrál. Így bizonyos tulajdonságokat (tehát pl. program specifikációt) betartó függvények létezésének belátását is ezekre korlátozza. A specifikáció konzisztenciájának bizonyításaként mintegy adódnia kell a specifikációt teljesítő értéknek. A klaszszikus specifikáció – implementáció – verifikáció jellegű ellenőrzést felváltja egy specifikáció – konzisztencia-ellenőrzés – implementáció-extrakció lépéssorozat, ahol az utolsó lépés automatikus.
Konstruktív nyelvre illetve környezetre példa a NUPRL rendszer.

Hibrid illetve széles spektrumú nyelvek: Azokat a nyelveket nevezhetjük széles spektrumú nyelveknek, amelyek a fejlesztési folyamat több (esetleg minden) fázisában, tehát a specifikáció, tervfinomítás, implementáció, verifikáció, tesztelés folyamán használhatóak. Ugyanakkor ebbe a kategóriába tartoznak azok a nyelvek is, amelyek a fenti ”tiszta” kategóriák elegyítésével állnak elő.
Ilyen nyelv például a LOTOS, amely a processz nyelvekből és az algebrai nyelvekből (absztrakt adattípusok) is tartalmaz elemeket.

A specifikációs nyelvek használatának egyik trendje, hogy a rendszer különböző aspektusait más és más, a célnak leginkább megfelelő nyelvvel írják le. A modell-orientált illetve algebrai nyelvek jól támogatják a komplex adatstruktúrák és a rajtuk értelmezett relációk, függvények leírását, ugyanakkor szekvenciális jellegű végrehajtást tételeznek fel. A processz leíró nyelvek konkurens rendszerek viselkedését adják jól meg (viselkedési szekvenciák, elérhetőségi fák, esemény sorrendezés fogalmait használva), de az adatstruktúrák általában nem interpretáltak (csak token jellegű adatáramlást vesznek figyelembe) illetve csak az egész számok tartományára korlátozottak.
A másik trend szerint több aspektust támogató, hibrid nyelveket kell definiálni és használni. Pl. az E-LOTOS javaslat a processz operátorok és algebrai adatstruktúrák mellett időbeliség kifejezését is támogatja.

6.2. Félformális módszerek

A félformális módszerek rendszerint formális szintaxissal (modell elemek és ezek lehetséges kapcsolatainak leírása) de informális szemantikával (a modell elemek és kapcsolatok jelentésének leírása) rendelkeznek.

Általános struktúra leírása: Funkcionális blokkdiagramok (általában megkötés nélküli ”doboz-diagram” a természetes leírás mellett, jobbára illusztrációs céllal).

Adatáramlás leírása: Adatáramlási diagramok használhatók, amelyek a következő elemekből állnak:

- csomópontok: adat-átalakító egységek; a bemeneti adatot kimenetté alakítja; az átalakítás módja magyarázatként szerepel; hierarchikusan finomítható.
- nyilak: adatáramlás; az adat meghatározása magyarázatként szerepel
- logikai operátorok: adatáramlás összefüggései

Vezérlési folyam leírása: Véges állapotú gépek (állapotátmeneti diagramok), szekvencia diagramok, időbeli Petri-hálók használhatók.

- Állapotátmeneti diagram: állapotok (hierarchiában is lehetnek), átmenetek (bemeneti eseménnyel és akcióval címkézve); teljesség, ellentmondásmentesség és elérhetőség szempontjából ellenőrizhető, tesztesetek generálhatók.
- Szekvencia diagram: egy lehetséges lefutás (forgatókönyv) megjelenítése.
- Petri-háló: konkurens rendszerfolyamatok leírása; univerzális (adat, kommunikáció és vezérlés leírása is; minden kapcsolatot struktúrába konvertál)
  - helyek: állapotok, feltételek, adatok tárolója
  - átmenetek: állapotátmenet, feltétel változása, adatfeldolgozás, szinkronizáció
  - tokenek (színezett is lehet): állapot aktív, feltétel igazsága, áramló adat

Logikai feltételek, összefüggések leírása: Igazságtáblázatok Boole típusú változók között (bemenet-kimenet kapcsolatok); automatikusan ellenőrizhetőek.

6.3. Strukturált metodika

A strukturált metodikák alapvető jellemzői a következők:

- a teljes rendszer meghatározása, beleértve a környezet megkötését is;
- nagyobb problémák kezelhető részekre osztása (a rendszer adatainak és funkcióinak felbontása) strukturált módon, logikai sorrendben;
- ellenőrzőlisták csatolása a meghatározást igénylő dolgokról;
- egyszerű, átlátható, intuitív, pragmatikus eljárások és jelölésrendszer;
- a támogató jelölések meghatározzák a problémát és a rendszerelemeket, de a feldolgozási funkciókat (viselkedést) általában informálisan adják meg.

Példák a strukturált metodikára:

- JSD (Jackson System Development): Meghatározza a rendszerfunkciókat a célkörnyezetben megvalósítható módon. A rendszermodell az összes folyamat strukturált leírásával bővül.
  - Modellezési fázis:
    - *akciók* (atomi külső események attribútumokkal, rendezéssel) és *entitások* (valós világbeli dolgok amiket az akciók befolyásolnak) azonosítása: mi történik és ez miket érint,
    - az akciók *sorrendezésének* meghatározása (szekvencia, választás, iteráció): entitás struktúra diagram.
    - *processzek* létrehozása az egyes entitásokhoz (JSP: Jackson Structured Programming).
  - Hálózati fázis: A rendszer működésének leírása *kommunikáló szekvenciális processzek hálózata* formájában; ez fokozatosan bővíthető (rendszerspecifikációs diagram). Új processz csatolható a többihez adatáramlással (üzenetküldés) vagy közvetlen állapotvektor hozzáféréssel.
  - Implementációs fázis: A futtató környezethez való illesztés (állományok, adatbázisok stb.).
- Valósidejű Yourdon (Ward-Mellor): Specifikációs és konstrukciós technika 3 fázisban:
  - alapvető modell: *környezeti modell* (határfelület és külső események) + *viselkedési modell* (eseményekre adott válaszként a rendszer által elvégzett transzformációk és adatok),
  - konstrukciós (implementációs) modell: az előírt viselkedési módokat megvalósító *folyamatok* (processzorokhoz rendelve és modulokra bontva),
  - szoftver és hardver megvalósítás.
- SADT (Structured Analysis and Design Technique): Központi szerepet játszik az aktivitás-faktor diagram. A *tevékenységek* dobozokba (action box) vannak csoportosítva, ezek más tevékenységi dobozokkal *faktor-relációkkal* (nyilak) vannak összekapcsolva. A dobozok és nyilak címkével (leírással) rendelkeznek (igék illetve főnevek). A faktorok négy típusa:
  - bemenet (bal oldalon): a dobozon belül manipulálható belépő dolgok (anyagi vagy nem anyagi)
  - vezérlés (doboz tetején): utasítás, választási feltétel a tevékenység végrehajtásához

  - mechanizmus (doboz alján): valamilyen erőforrás (személyzet, berendezés) ami a tevékenységhez szükséges
  - kimenet (jobb oldal): tevékenység eredménye

  A tevékenységi dobozok hierarchikusan felbonthatók másodlagos tevékenységi dobozokra és relációkra (fa struktúrájú diagramok). A sok faktorral erősen összekapcsolt tevékenységek egy dobozba tartoznak.

6.4. Követelményspecifikáció készítésének támogatása

Az automatikus követelménykezelők következő feladatai (funkciói) különíthetők el:

- Bemeneti követelmények (követelménylisták) nyilvántartása: A belső tárolás mellett szükséges lehet külső dokumentumokra is hivatkozni (pl. szabvány szövegek), de ez esetben nem garantált ezek változásainak kezelése. Célszerű a hierarchikus felépítés.
- A követelmények változásainak kezelése: A logikai struktúra mellett az időbeli struktúrát is kezelni kell. Az egyes állapotok elmenthetősége mellett az egyes lépéseket is vissza kell tudni játszani.
- Kapcsolatok nyilvántartása a követelmények valamint a modell elemek és kódrészletek között. A kapcsolatok változását is olyan módon kell kezelni, mint a követelményekét.
- Navigáció a kapcsolatok mentén mind a követelmények, mind pedig a megvalósítás (modell elemek és kódrészletek) irányából: Ez alapján lehet meghatározni, hogy egy változás milyen elemeket érint.
- Fedettségi listák készítése: Ezek kimutatják, mely követelmények nincsenek lefedve, (nincs megvalósító kódrészlet), mely kódrészletek tűnnek feleslegesnek (nincs hozzárendelt követelmény). Az elemzést segíti, ha ezek a listák grafikusan is megjeleníthetők, illetve ezekből formázott jelentés készíthető.
- Jogosultságok kezelése a követelmények illetve kapcsolatok beviteléhez és megváltoztatásához: A feladatokat (munkaköröket) és a hozzá szükséges jogokat felhasználók, fejlesztők illetve ezek csoportjai esetén szükséges definiálni.
- Értesítési rendszer a követelmények illetve kapcsolatok változásának jelentésére adott fejlesztői csoportok felé. Egy bekövetkező változásról tudnia kell annak, akinek a munkáját érinti.
- Biztonsági megoldások a dokumentumok sértetlenségének, megbízhatóságának és letagadhatatlanságának érdekében. Ez elsősorban a megrendelõk és a fejlesztők közötti viták esetén lehet fontos.

Példa: Rational RequisitePro

6.5. Követelmény-specifikáció ellenőrzése

A követelmény-specifikáció verifikációja szempontjából az ellenőrizendő tulajdonságok taxonómiája a következő:

- Teljesség: Minden szükséges részlet teljes pontossággal szerepel, azaz nincs
  - kihagyott, kifejtésre vagy pontosításra váró részlet (TBD, ”to be determined” bejegyzések).

  - nemlétező hivatkozás (dokumentumokra, szabványokra, jegyzőkönyvekre).
  - hiányzó specifikációs részlet (pl. interfész-leírás, használati feltételek).
  - hiányzó funkció (pl. hibakezelés).
  - hiányzó eszköz megadás (pl. teszt eszköz, utófeldolgozó).

- Konzisztencia: Nincsenek ellentmondó követelmények.

  - Belső konzisztencia: Nincs ellentmondó kitétel.
  - Külső konzisztencia: Nincs ellentmondás külső dokumentumokkal, feltételekkel.
  - Követhetőség: Nincs előzmény nélküli, ”kóbor” követelmény illetve döntés (elsősorban későbbi tervdokumentumokban).

- Megvalósíthatóság: Kockázatok felismerése.

  - Emberi használhatóság: A specifikált rendszer kielégít-e (különböző szintű) elvárásokat.
  - Erőforrástervezés: Megengedhető költségek mellett megvalósítható illetve üzemeltethető-e a specifikált rendszer.
  - Programtervezés: Karbantarthatóság, hordozhatóság, megbízhatóság biztosítható-e.
  - Kockázatok: Technikai (adatbiztonság, teljesítmény, ember-gép kapcsolat), költség (platform, üzemeltető), környezeti (felhasználó, bemeneti adatok, külső interfészek) szempontból ismert-e minden körülmény. Ezek félreértése, nem ismerete kudarcot eredményezhet.

- Tesztelhetőség: Megvalósítható technikák megadhatók annak ellenőrzésére, hogy a követelmények teljesülnek-e. Jellegzetesen nem tesztelhető követelmények: legyen ”modern megoldás”, ”optimalizált erőforráshasználat”, ”valósidejű működés”.

  - Specifikus: Egyszerre egy (vagy meghatározott körű) követelményre koncentrál.
  - Egyértelmű: A siker vagy kudarc eldönthető.
  - Számszerűsíthető: Amennyire lehet, a teszt eredmények számszerűen kiértékelhetők.

Kisebb rendszerektől nagyobb rendszerek specifikációi felé haladva a következők javasoltak:

- Hivatkozások, követhetőség automatikus ellenőrzése.
- Modell alapú (formális) viselkedési specifikáció és ennek automatikus ellenőrzése.
- Felhasználói felületek forgatókönyv-alapú átvizsgálása.
- A magas kockázatú elemek vizsgálata gyors prototípus készítésével.

A specifikáció ellenőrzésének lépései a következők:

- Információgyűjtés (alkalmazási terület szakértőitől, akik a kritikus pontokat ismerik).
- Az informális követelmények ellenőrzése (pl. átolvasás és egyetértés).
- Az informális követelmények formalizálásának ellenőrzése (pl. hiba az okozat használata az ok helyett).
- A formális követelmények ellenőrzése (pl. szimbolikus végrehajtással, szimulációval).

6.6. Teszt specifikáció ellenőrzése

A teszt specifikáció az egyes követelmények tesztelését kell rögzítse (lásd az előbb előírt tesztelhetőségi tulajdonságokat), elsősorban a validációs fázisra gondolva. Ellenőrizni kell, hogy a teszt specifikáció választ ad-e minden követelmény esetén a következő kérdésekre:

- Ki hajtja végre a tesztet?
- Mikor kell a tesztet végrehajtani?
- Milyen tesztelési technikát kell alkalmazni?
- Mi a siker/kudarc ismérve?

7. Architektúra tervezés

Szoftver architektúra: Szoftver elemek + tulajdonságaik + kapcsolataik. (Több struktúra is megadható különféle szempontok szerint, pl. funkcionális, megbízhatósági illetve teljesítmény szempontból. A kapcsolatok is ennek megfelelően változhatnak, pl. funkció hívások, szolgáltatásnyújtás és erőforrás-használat, hibaterjedés.)
Cél: Olyan szoftver-architektúra kifejlesztése, amely a szoftver-biztonságintegritási szint (BI) által megkövetelt mértékben eleget tesz a specifikáció előírásainak. A hardver-szoftver kölcsönhatás meghatározása.
Bemenet:

- Rendszer biztonsági követelmények specifikációja és architektúra terv.
- Szoftverkövetelmény-specifikáció.
- Szoftver minőségbiztosítási terv.

Kimenet:

- Szoftver architektúra specifikáció.
- Szoftver-hardver integrációs teszt terv.
- Szoftver architektúra verifikációs jelentés.

A szoftver architektúra minőségi jellemzői és a kapcsolódó, az architektúrát meghatározó tervezői döntések:

- Teljesítmény: Erőforrás menedzsment, erőforrás birtoklás.
- Megbízhatóság és rendelkezésre állás: Hibadetektálás, hibabehatárolás, helyreállítás, hibakezelés.
- Biztonságosság: Veszély kiküszöbölés, veszély csökkentés, veszély kontroll, kárcsökkentés.
- Adatbiztonság: Támadás kivédés, támadás felderítés, helyreállítás.
- Tesztelhetőség: Vezérelhetőség, megfigyelhetőség.

- Módosíthatóság és hordozhatóság: Módosítási igény lokalizálás, hullámhatás elkerülése, késői kötés (futásidőbeli).
- Használhatóság: Felhasználói felület elkülönítés (pl. MVC), adaptivitás felhasználói modell alapján.

Az architektúra verifikáció általános jellemzői:

- A szoftver elemek azonosítása a V&V szempontjából: új, már meglévő, már validált stb. komponensek.
- COTS modulok alkalmazása a különböző BI szinteken:
  - 0. szinten: további feltételek nélkül befogadható;
  - 1. és 2. szinten: validálni szükséges;
  - 3. és 4. szinten: meghibásodásokat és hatásukat elemezni kell; hiba elemzési és védelmi stratégia szükséges; verifikációs tesztelés előírt; validáció a védelmi mechanizmusra is ki kell terjedjen.
- Verifikáció és validáció a legmagasabb előírt biztopnságintegritási szintű modulnak megfelelően kell történjen (kivéve, ha bizonyítható, hogy az alacsonyabb szintű modul hibája nem veszélyezteti a magasabb BI szintű modulok működését).
- Szoftverhiba hatáselemzés (SEEA - Software Error and Effects Analysis): Független szakember(ek) által végzett *biztonságkritikus modul azonosítás*, ez alapján *szisztematikus elemzés* a megfontolandó hibákról, a hibadetektálásról, a bennmaradó kritikus hibákról, valamint *szintézis*, azaz a nem biztonságos esetekhez előírt megoldások, verifikáció és validáció meghatározása.
- Hibaelkerülés és hibakezelés együttesen (kiegyenlítetten) alkalmazandó.

7.1. A hibaelkerülés módszerei (analízis)

A *szisztematikus elemzés* módszerei:

- Hibafa elemzés: Olyan eseménykombinációk meghatározása, amelyek rendszerszintű hibajelenséghez vagy veszélyes állapothoz vezetnek.
- Eseményfa elemzés: Az egyes eseményszekvenciákat (forgatókönyveket) adja meg az egymás után bekövetkező események szisztematikus felsorolásával (minden eseményre elágazásként felvehető, hogy bekövetkezik-e vagy nem). A forgatókönyvek rendszer szinten jellemezhetők (veszélyesek-e vagy nem).
- Ok-okozati diagramok: A hátralépő (okokat elemző) hibafa analízist egyesíti az előrelépő eseményfa analízissel, mivel a forgatókönyvek mentén az egyes események bekövetkezéséhez hibafa hozzárendelésével adja meg az okokat.

Az *architektúra modellezése és a modell alapú analízis* célja elsősorban az architektúra-változatok összehasonlítása és az érzékenységvizsgálat (hogyan függ a rendszerszintű jellemző a komponens jellemzőtől).

- Teljesítmény-modellezés: Modellalkotás annak ellenőrzésére, hogy a teljesítmény és időzítés jellegű követelmények az adott erőforrásokkal teljesíthetők-e (van-e szűk keresztmetszet, telítődés). Vizsgálandók az átlagos értékek illetve a legrosszabb eseti (worst case) értékek (utóbbiak szigorúan valósidejű rendszerek esetén). A részletes modellezés előtt egy egyszerű igényösszegzéssel durván ellenőrizhető a követelmények betarthatósága.

- Megbízhatósági modellezés: Annak modell alapú vizsgálata, hogy az egyes (hardver és szoftver) komponensek hibái mikor vezetnek rendszerszintű hibához. A modell megoldása (analitikusan vagy szimulációval) a rendszerszintű megbízhatósági jellemzőket adja, ennek komponens paraméterekre való érzékenységének elemzésével a szűk keresztmetszetek illetve a szükséges redundancia mértéke meghatározható.

- Biztonságosság modellezése: A komponens szintű lokális veszélyek azonosítása és annak vizsgálata, ezek milyen módon (kombináció illetve forgatókönyv esetén) vezethetnek rendszerszintű veszélyhez.

A modellezéshez tipikusan használt alapadatok és a kiszámítható rendszerszintű jellemzők:

	Teljesítmény modell	Megbízhatósági modell	Biztonsági modell
Komponens jellemzők	processzor ütemezés taszk prioritás taszk aktiválási gyakoriság funkció végrehajtási idő funkció fázisok	meghibásodási tényező hiba lappangási idő javítási tényező	veszély gyakoriság
Kapcsolati jellemzők	hívás továbbítási gyakoriság hívás szinkronitása	hibaterjedési valsz. hibaterjedés logikája javítási stratégia	veszélyek kombinációja veszély forgatókönyv
Rendszer jellemzők (számított)	funkció kiszolgálási idő taszk áteresztőképesség processzor kihasználtság hívás késleltetés	megbízhatóság rendelkezésre állás készenlét MTTF, MTTR, MTBF	veszély gyakoriság

7.2. A hibakezelés módszerei (szintézis)

A tervezési időben nem elkerülhető hibák kezelését a futásidőbeli hibadetektálás és a hibatűrés (redundancia) biztosítja.

- A futásidőbeli hibadetektálás módszerei:

  – Defenzív programozás: Abnormális adatáramlás (típus, értéktartomány, hihetőség ellenőrzése) és vezérlési folyam (utasítás-szekvencia) felismerése és erre reagálás.

  – Meghibásodásbizonyító (failure assertion) programozás: Beépített állítások használata, amelyek futásidőben jelzik az invariánsok, elő- illetve utófeltételek megsértését.

  – Biztonsági zsák (safety bag) módszerek: A biztonsági zsák egy eltérő tervezésű, külső, független számítógépre telepített monitor. Kizárólagos feladata a biztonságos (bár nem feltétlenül helyes) működés ellenőrzése, és a nem biztonságos állapotba lépés megakadályozása (visszakényszerítés biztonságos állapotba).

  - Megvalósított esetek tárolása: Nem engedélyezett végrehajtási útvonalak felismerése annak alapján, hogy az engedélyezett útvonalakat referenciaként tárolják. Az útvonalak döntéseket, hozzáférési szekvenciákat tárolhatnak prímkódolás, dominó, szomszédossági mátrix stb. alapján.

- Redundancia alkalmazása:

  - Hardver redundancia: Hibafelfedés és hibadiagnózis. Az általában azonos redundáns elemek segítségével tranziens és állandósult működési (de nem tervezési) hibák detektálása, diagnosztika után hibakezelés.
  - Szoftver redundancia: Tervezési hibák elleni védekezés is eltérő tervezéssel.
    * Diverz programozás (N-version programming). Hiba maszkolás a felhasználás előtt. Ugyanazon specifikáció alapján többféle eltérő megvalósítás (tervezők, algoritmus, nyelv stb.), szavazás (n egyezés, n/k egyezés, többségi szavazás) és beavatkozás (állandósult hibával rendelkező kimenet lekapcsolása) illetve hibaelfedés szavazással.
    * Helyreállító blokk (recovery blokk): Több blokk végrehajtása egy elfogadhatósági teszt eredményétől függően.
  - Információ redundancia: Hibafelismerő kódok. Sérülékeny információ kiegészítése redundáns bitekkel a sérülés detektálása illetve javítása érdekében.
  - Idő redundancia: Újrapróbáló hibajavítás. Ismételt végrehajtás tranziens hardver illetve kommunikációs hibák elfedésére (retry, restart, reboot).

- Nem ajánlott módszerek biztonságkritikus rendszerekben:

  - Visszalépéses helyreállítás (idő- és tárigény valamint konzisztencia problémák elosztott rendszerekben).
  - Előrelépéses helyreállítás (szelektív korrekció alkalmazás-specifikus, hibafedéstől és kárfelméréstől függő).
  - Mesterséges intelligencia alapú hibajavítás (nehezen verifikálhatók a heurisztikus módszerek).
  - Dinamikus szoftver-rekonfiguráció (nehezen verifikálható minden lehetséges eset).

8. Részletes tervezés

Cél a részletes szoftver konstrukció kialakítása az architektúrának megfelelően, valamint eszközkészlet választás a tervezéshez (pl. nyelv, fordítóprogram, tesztelő eszközök). Nagy rendszerek esetén az áttekinthetőség érdekében különválasztják a teljes *szoftver konstrukció* és a *modul konstrukció* fázisát.
Bemenet:

- Szoftverkövetelmény-specifikáció, szoftver architektúra specifikáció.
- Szoftver minőségbiztosítási terv.

Kimenet a szoftver konstrukció fázisában:

- Szoftver konstrukció specifikáció:
  A *szoftver konstrukció* határozza meg a modulok közötti interfészeket, adatstruktúrákat, rendszerszintű algoritmusokat (modulok együttműködése). A használt nyelvnek támogatnia kell a következőket:
  - Absztrakció és modularitás.
  - Információáramlás, sorrendiség, időbeliség leírása.
  - Adatstruktúrák meghatározása.
  - Érthetőség, verifikálhatóság.
  - Karbantarthatóság és újrafelhasználhatóság (modularitás, információrejtés).
  - A választott programozási nyelvek és eszközök követelményeit lásd a modul implementációs fázisban!
- Szoftver integrációs teszt terv.
  A *teszt tervekhez* használatos eljárásokról és módszerekről ld. a tesztelés életciklus fázisokat!
- Szoftver architektúra és konstrukció verifikációs jelentés.
  A *szoftver architektúra és konstrukció verifikáció* az alábbiakra irányul:
  - Megfelelőség a követelmény-specifikáció - architektúra terv - konstrukció specifikáció között.
  - Teljesség, ellentmondásmentesség, megvalósíthatóság, tesztelhetőség (ld. a követelmény-specifikáció ellenőrzése fejezetben).
  - Az elkészült integrációs teszt tervek megfelelnek-e az architektúra és konstrukció terveinek.
  - A teljesítmény és időzítés jellegű követelmények meghatározása megtörtént-e (sikeres teljesítés kritériumai, mérések pontossága, végrehajthatóság minél korábbi fejlesztési fázisban).

Kimenetek a modul konstrukció fázisában:

- Modultervezési specifikáció.
  A *modultervek* határozzák meg az alacsonyabb szintű komponenseket (függvények), belső interfészeket, részletes algoritmusokat és adatstruktúrákat. A használt nyelvekre vonatkozó követelmények hasonlóak, itt a viselkedés leírása nagyobb szerepet kap.
- Modul teszt specifikáció:
  A *teszt tervekhez* használatos eljárásokról és módszerekről ld. a tesztelés életciklus fázisokat leíró fejezeteket!
- Szoftver modul verifikációs jelentés.
  A *szoftver modul verifikáció* a következőkre irányul:
  - Megfelelőség a konstrukció specifikáció - modultervek között.
  - Teljesség, ellentmondásmentesség, megvalósíthatóság, tesztelhetőség vizsgálata.
  - Az elkészült modul teszt tervek megfelelnek-e a modul terveknek, a tesztjelentés formátuma rögzített-e.

8.1. Módszerek a szoftver és a modul konstrukcióhoz

- Formális - félformális - strukturált módszerek a részletes tervezéshez: ld. a követelmény-specifikáció fejezetben!
- A szoftver konstrukcióhoz szükséges a *moduláris megközelítés*.
  - Modulméret korlátozás: Megvalósító kód mérete korlátozandó.
  - Információrejtés: Globális változók kerülése, hozzáférések ellenőrzött függvények által.
  - Paraméter mennyiségi korlátozás: Kapcsolatok, paraméterek száma.
  - Egybemenetű és egykimenetű függvények: Belépési-kilépési pontok számának korlátozása.
  - Teljesen meghatározott interfészek (szükséges feltétel).

8.2. Módszerek a terv verifikációhoz

A tervek verifikációjához alkalmazható módszerek:

- Formális bizonyítás:
  - Algoritmusok helyességének bizonyítása: Modell ellenőrzés, tételbizonyítás.
  - Finomítás helyességének bizonyítása: Ekvivalencia ellenőrzés, tételbizonyítás.
- Statikus elemzés: Ezek mind az algoritmus és adatstruktúra modellen, mind a forráskódon (ld. később) elvégezhetők.
  - Ellenőrzőlisták: Kérdések (ökölszabályok) összeállítása és ezek ellenőrzése. Informális, biztosítja, hogy a már kiderült ökölszabályokat ne hagyják ki, ugyanakkor a szabálykészlet teljessége nem bizonyítható, így hamis biztonságérzetet adhat.
  - Hibabecslés: Tapasztalatok alapján tipikus hibalehetőségek és kombinációk vizsgálata.
  - Határérték-elemzés: A paraméterhatárok kezelésének ellenőrzése az interfészeken illetve az algoritmusokban. Ezek alapján tesztek készítése (ld. ott).
  - Vezérlési folyam ellenőrzés: Algoritmus specifikáció vezérlési folyamának elenőrzése struktúráltság szempontjából (gráfredukciókkal egy csomóponttá redukálható-e a vezérlési folyam, ld. strukturált programozás alapkonstrukcióinak alkalmazása megvalósul-e).
  - Adatáramlási elemzés: Változók hozzáférési sorrendjének vizsgálata: olvasás írás előtt, nem használt, de specifikált változók.
  - Alattomos komponensek elemzése: Nemkívánatos információáramlási útvonalak és folyamatok felderítése olyan rendszerekben, ahol bizonyos (látens) feltételek nemkívánatos funkciókat indítanak el, vagy megkövetelt funkciókat akadályoznak. Pl. nem kívánt adatáramlás védett és védtelen komponensek között; időzítéssel kapcsolatos befolyásolás vagy járulékos információszerzés; félreérthető visszajelzések, amelyek beavatkozást vonnak maguk után; operátori beavatkozás félreérthető címkézése. Alapvetően a hardver-szoftver topológiai minták alapján végezhető, és az adatbiztonság szempontjából kritikus rendszerekben különös jelentőséget kap.
  - Szimbolikus végrehajtás. A specifikációnak való megfelelőség vizsgálata a részletes terv illetve modell (itt általában kézi) ”végrehajtásával”: Az egyes lépésekben a baloldal jobboldallal való helyettesítése, az elágazások és hurkok Boole-kifejezésekké alakítása, a változók végértékének követése.

- Dinamikus elemzés:
  - Prototípus készítés és animáció: A szoftver funkciók kockázatos illetve nehezen vizsgálható részének gyors megvalósítása és ez alapján történő ellenőrzése. Itt még nem veszik figyelembe a megvalósításra a későbbiekben vonatkozó előírásokat (pl. programnyelv, karbantarthatóság stb.).
  - Szimuláció: A végrehajtható modell illetve a prototípus interaktív futtatása és a tulajdonságok ellenőrzése. Olyan bemeneteket célszerű használni, amik a tényleges működés közben előfordulnak.
- Követhetőségi vizsgálat:
  - A követelmények követhetősége a tervre vonatkozó feljegyzésekkel minden tervezési fázisban (*követhetőségi mátrix*: követelmények és tervrészletek összerendelése). Listázhatók a nem lefedett követelmények, illetve a követelményhez nem köthető terv részletek.
  - A tervezői döntések követhetősége a hátterüket szolgáló információk megadásával (*tervezői döntés adatbázis*: döntések összerendelése a követelményekkel, tudásbázissal, analízis eredményekkel, kockázatelemzéssel és egyéb indoklással). Utólag is kideríthető, mi volt az adott döntés indoka illetve előfeltétele.
- Tesztek felülvizsgálata: Teljesség ellenőrzés (minden, a modultervben felírt követelményhez készült teszt).

Az ellenőrzés szervezésére vonatkozó ajánlások:

- Revízió/tervátvizsgálás (lásd. *felülvizsgálat* és *egyenrangú átvizsgálás* általános módszerei).
- Fagan-felülvizsgálat: Az informális ellenőrzés öt fázisból álló folyamata. A fázsiok a következők: ellenőrzés tervezése, előkészítése, ellenőrzés, átdolgozás a talált hibák javítására, a javítások követése.

9. Modul implementáció

Célok: Forráskód és támogató dokumentáció elkészítése valamint a forráskód (statikus, ”átolvasáson” és nem végrehajtáson alapuló) verifikációja.
Bemenet:

- Modultervezési specifikáció.

Kimenet:

- Forráskód és támogató dokumentáció.
- Forráskód verifikációs jelentés.

Előírások a forráskód készítéshez (a szoftver minőségbiztosítási terv része, illetve a szoftver konstrukció fázisában pontosítható):

- A követelmények követhetőségét a megvalósításig biztosítani kell.

- Eszközökre vonatkozó előírások:
  - A fordítóprogram szabvány szerinti elfogadott vagy tanúsított kell legyen.
  - Tanúsított eszközök általában csak a fordítók vagy kódgenerátorok (szabványhoz való illeszkedést bevizsgálták); egyéb eszközöknél is elvárt azonban a fejlesztés közbeni minőségbiztosítás. Olyan eszközt tilos használni, ahol nincs meglévő üzemi tapasztalat (validált vagy ”gyakorlatban bevált” eszköz).
- Programozási nyelvre vonatkozó előírások: A nyelvnek támogatnia kell a programhibák felismerését és illeszkednie kell a kiválasztott tervezési és tesztelési módszerekhez.
  - Strukturált programozás: Egyszerűen követhető vezérlési szerkezetek (elágazás, iteráció, függvényhívás) még a ”hatékonyság” rovására is. BASIC, PLM, assembler nem ajánlott. Az OO programozás ajánlott módszer.
  - Elemezhető programok: Strukturált programozás szabályai, egyszerű döntési feltételek.
  - Erősen tipizált programnyelv: Típusellenőrzést (pl. hívási argumentumokra is) a fordítóprogram támogatja.
- Tervezési és kódolási szabályok: Ez meghatározza a helyes programozási gyakorlatot, megtiltja a nem biztonságos nyelvi jellemzőket (nyelvi részhalmazt definiál) és leírja a forráskód dokumentálásának eljárásait.
  - Kódolási mód irányelveket rögzít (pl. interfészek formája, modulméretek, megjegyzések formája).
  - Adott programnyelvi konstrukciók kitilthatók (pl. goto, rekurzió, feltétel nélküli ugrás, mutató használat, automatikus típuskonverzió).
  - OO módszereknél korlátozhatók egyes konstrukciók (pl. egymásba ágyazás, polimorfizmus, többszörös öröklődés).
  - Dinamikus konstrukciók kitilthatók (objektumok dinamikus létrehozása, dinamikus változók).
  - Előírt mértékek a program összetettségére (ld. a módszereknél).

Alkalmazható módszerek a forráskód verifikációhoz:

- Statikus elemzés: Kód szinten elvégezhető:
  - Fenti előírások teljesülésének vizsgálata (követhetőség, eszközök, nyelv, kódolási szabályok betartása).
  - Az előző alfejezetben a részletes tervezés szintjén ajánlott módszerek (ellenőrzőlisták, hibabecslés, határérték-elemzés, vezérlési folyam ellenőrzés, adatáramlási elemzés, alattomos komponensek elemzése, szimbolikus végrehajtás).
- Dinamikus elemzés és tesztelés: Ez már a modul tesztelés feladata, ld. a következő alfejezetben.
- Mértékek: A programok jellemzőinek előrejelzése magából a forráskód tulajdonságaiból (és nem a fejlesztés történetéből). Jellemzően a strukturális összetettség vizsgálható automatikusan:

  - Gráfelméleti komplexitás (ciklomatikus számok): Komplex modul nehezebben tekinthető át illetve nehezebben tesztelhető.
  - Aktiválási módok száma (hozzáférhetőség): Minél inkább hozzáférhető, annál könnyebben kiszűrhetők a hibák.
  - Programhosszúság (operátorok és operandusok száma): Áttekinthetőség és tesztelhetőség könnyebb a rövidebb modulok esetén.
  - Be- és kimenetek száma: Ezek minimalizálása célszerű.

- Szoftverhiba hatáselemzés (SEEA): Független szakemberek által végzett ellenőrzési módszer (ld. előbb).

10. Modul tesztelés

Célok: Kimutatni, hogy a modul a specifikációnak megfelelően működik.

- Dijkstra: A tesztelés a hibák *jelenlétét*, és nem a hibamentességet tudja megmutatni!
- Hoare: Elméletileg a tesztelés a helyesség egy *induktív bizonyításának része*: Ha a program egy adott (teszt)esetre jól működik, akkor várhatóan más esetekre is jól működik.

Bemenet:

- Modultervezési specifikáció.
- Modul teszt specifikáció.

Kimenet:

- Szoftvermodul teszt jelentés.
  - A szoftvermodul tesztelést a tervező-fejlesztő végezheti.
  - A szoftvermodul tesztjelentés rögzíti a teszteseteket és azok eredményeit gépi formában, valamint a megállapítást az elfogadásról vagy elutasításról.
  - A tesztjelentésben rögzíteni kell, hogy minden forráskód-utasítást legalább egyszer elvégeztek.

Az alkalmazható módszereket részletezik a következő alfejezetek.

10.1. Funkcionális (fekete doboz) tesztelés

A funkcionális tesztelés a modul belsejére nem koncentrál, hanem a kívülről megfigyelhető funkciókat teszteli. A tesztesetek megvalósítása az alábbiak alapján történik:

- Ekvivalencia osztályok és bemeneti adatfelosztás: A modul bemeneti tartományát intervallumokra osztják. Az intervallumokat a funkcionális specifikáció alapján kell elkülöníteni, bemenet- vagy kimenet-orientáltan. Úgy kell az elkülönítést végezni, hogy az egyes intervallumokban a szoftver azonos viselkedését fedjék le. Az intervallumokból egy-egy reprezentatív tesztet választanak, azt feltételezve, hogy ha erre hibátlanul lefut a teszt, akkor az intervallum többi elemére is.

  - Az intervallumok és a tesztek meghatározása többnyire heurisztikus folyamat. Példák: használati esetek, különféle üzemmódokhoz tartozó bemenetek.
  - Cél, hogy egy teszt minél több bemeneti feltételt írjon elő (minél több hibát felfedhessen), és minél több bemenettel legyen ekvivalens (minél kevesebb tesztet hajtsunk végre).

- Határérték elemzés: Az egyes intervallumok határai közelében választanak teszt bemeneteket (szélső értékek, mint nulla, üres karakterlánc, üres lista; kezdő- és végértékek).

  - Egy határérték általában 3 tesztet jelent (kicsit kisebb, éppen a határon, kicsit nagyobb érték), így egy intervallum 5-7 tesztet jelent (két határérték és egy belső reprezentatív elem).

- Ok-okozati kapcsolatok: A bemenetek és kimenetek intervallumai közötti logikai kapcsolatokat (implikációkat) kell felvenni (ok: bemenet, okozat: kimenet), és ezeket a kapcsolatokat tesztelni.

  - Boole-gráf: Bemeneti és kimeneti intervallumok összekapcsolása, a meg nem engedett kombinációk bejelölésével.
  - Döntési táblázat: A kapcsolatok igazságtáblaként való felírása; minden logikai kombinációt le kell fedni a tesztelés során.

- Állapot-átmenet tesztelés: Ha a specifikáció állapotgép alakjában állt rendelkezésre, akkor annak trigger eseményeit kell bemenetként biztosítani és az akciókat mint kimeneteket vizsgálni.

- Interfész tesztelés: A modul interfészek szisztematikus tesztelése.

  - Az egyes interfészeken értelmezett intervallumok kombinációit kell tesztelni kimerítő módon (ez csak egyszerű moduloknál lehetséges) illetve a reprezentatív értékek kombinálásával.
  - Fontos a nem elfogadható értékek tesztelése. Tipikusan a nem elfogadható értékeket egyenként, normál értékekkel kombinálva tesztelik, hogy az egymást kioltó hatások ne zavarják meg a teszt eredmények értékelését.

- Véletlen tesztelés: Véletlen tesztvektorok generálása; kis számítási teljesítményt igényel, gyors, de hibafedése kicsi. A referencia válasz számítható, szimulálható vagy csak elfogadhatósági vizsgálat történik. Kiegészítő módszerként használatos.

- Folyamatszimuláció: Szimulálják a modul környezetét (pl. a vezérelt rendszert, hardvert) annak érdekében, hogy a modul viselkedését vizsgálni tudják.

  - A szimulációs környezetnek minden olyan bemenetet biztosítania kell, ami a valós rendszerben meglesz; a modul kimenő jeleire úgy kell reagálnia, mint a valós rendszer.

10.2. Strukturális (fehér doboz, üvegdoboz) tesztelés

A strukturális tesztelés során a modul belsejét is szem előtt tartva történik a tesztek tervezése. Ez gyakorlatilag csak modul szinten történhet meg. Cél a belső működés végigkövetése. A tesztesetek megvalósítása az alábbiak alapján történik:

- Struktúra alapú: A programstruktúra egyes részeinek (utasítások, elágazások, feltételek, adatáramlás, hívógráfok, útvonalak) részletes tesztelése. Jól kezelhető modellje a program vezérlési gráf illetve adatfolyam gráf. Fedettségi kritériumokat lehet ezekhez a tesztekhez definiálni.
  Vezérlési gráf alapú kritériumok:

  - Utasítás fedettség: Minden utasítás végrehajtása. Nem szigorú, pl. utasítások kihagyási feltételeit nem veszi figyelembe. Általában előírás a 100%-os utasítás fedettség.
  - Döntési ág fedettség: Minden döntési ág végrehajtása. Nem szigorú, mert a végrehajtás sok nyelvben (pl. C esetén || illetve $\&\&$) optimalizált, így feltételek kiértékelése kimaradhat.
  - Feltétel fedettség: Minden kombinációt tesztelni a döntési feltételekben. Aránylag bonyolult a sok lehetséges kombináció miatt.
  - Útvonal fedettség: Minden független út bejárása. A független utak maximális száma, ha a vezérlési gráf csúcsainak száma N, éleinek száma E: $CK = E - N + 2$. A 100% útvonal lefedettség biztosítja a 100% utasítás és 100% döntési ág fedettséget is. Probléma: A gráfelméleti módszerekkel kiszámolt útvonal nem biztos, hogy bejárható (ellentmondó teszt adatokhoz vezethet).

  Adatfolyam gráf alapú kritériumok: Jelöljük egy *v* változó értékadását *def v*-vel, számításban való felhasználását *c-use v*-vel és feltételes elágazás feltételében való használatát *p-use v*-vel. *Def-clear v* útvonal egy olyan útvonal a vezérlési gráfban, amiben nincs *def v* címkéjű utasítás, *d-u v* útvonal egy olyan útvonal, amiben *def v* címkéjű az első csomópont, *p-use v* vagy *c-use v* címkéjű az utolsó csomópont, közben *def-clear v* útvonal van, valamint nincs benne hurok (legfeljebb a teljes *d-u v* útvonal képez egy hurkot). A tesztelés célja az, hogy minden értékadás (*def*) és későbbi felhasználás (*use*, azaz *c-use* vagy *p-use*) kapcsolatát letesztelje. A következő kritériumokat használják a leggyakrabban:

  - all-defs: $\forall v, \forall def\ v$: egy *use v* címkéjű utasításig legalább egy *def-clear v* útvonal tesztelése.
  - all-p-uses: $\forall v, \forall def\ v$: minden *p-use v*-ig legalább egy *def-clear v* útvonal tesztelése.
  - all-c-uses: $\forall v, \forall def\ v$: minden *c-use v*-ig legalább egy *def-clear v* útvonal tesztelése.
  - all-p-uses/some-c-uses: $\forall v, \forall def\ v$: minden *p-use v*-ig és legalább egy *c-use v*-ig (ha nincs *p-use v*) legalább egy *def-clear v* útvonal tesztelése.
  - all-c-uses/some-p-uses: $\forall v, \forall def\ v$: minden *c-use v*-ig és legalább egy *p-use v*-ig (ha nincs *c-use v*) legalább egy *def-clear v* útvonal tesztelése.
  - all-uses: $\forall v, \forall def\ v$: minden *use v*-ig (tehát minden *p-use v*-ig és minden *c-use v*-ig) legalább egy *def-clear v* útvonal tesztelése. Ez lefedi az eddigi kritériumokat is.
  - all-paths: $\forall v, \forall def\ v$: minden *use v*-ig minden bejárható *def-clear v* útvonal tesztelése. Ez lefedi az eddigi kritériumokat is. A kritérium teljesítése problémás abban az esetben, ha hurkok (ciklusok) vannak az útvonal mentén, és ezek tetszőleges (előre nem ismert) sokszor bejárhatók.
  - all-du-paths: $\forall v, \forall def\ v$: minden bejárható *d-u v* útvonal tesztelése minden *use v*-ig. Mivel a *d-u v* útvonal hurokmentes, vagy egyszerű hurok, ezért a hurok-bejárást a kritérium nem írja elő. Ugyanakkor ez a kritérium nem fedi le az előző kritériumokat.

- Hibabecslés: A tesztelési tapasztalatok és ”megérzések” alapján a szisztematikusan generált tesztekhez újabbak adhatók.

- Hibabeültetés és hibakeresés: A teszteset-gyűjtemény megfelelősége úgy vizsgálható, hogy a programba hibákat helyeznek el (pl. mutációs kódgenerátorral), és ezek felfedését ellenőrzik. A *megtalált beültetett hibák* és az *összes beültetett hiba* aránya becslője a *megtalált valódi hibák* és az *összes valódi hiba* arányának.

10.3. Valószínűségi tesztelés

A valószínűségi tesztelés során a talált hibák számából és időbeli eloszlásából vonhatók le következtetések a tesztelt modul megbízhatóságára (pl. adott követelményre vonatkozó meghibásodási valószínűség, időegységre vonatkozó meghibásodási valószínűség, hibatartalom valószínűsége) vonatkozóan. A tesztek szisztematikus illetve véletlen tesztek is lehetnek, ezek futtatása és a teszt eredmények kiértékelése általában automatizáltan történik. Az eredményekből a kívánt valószínűségi értékek különféle modellek (reliability growth models) alapján számíthatók ki.

10.4. Eszközök

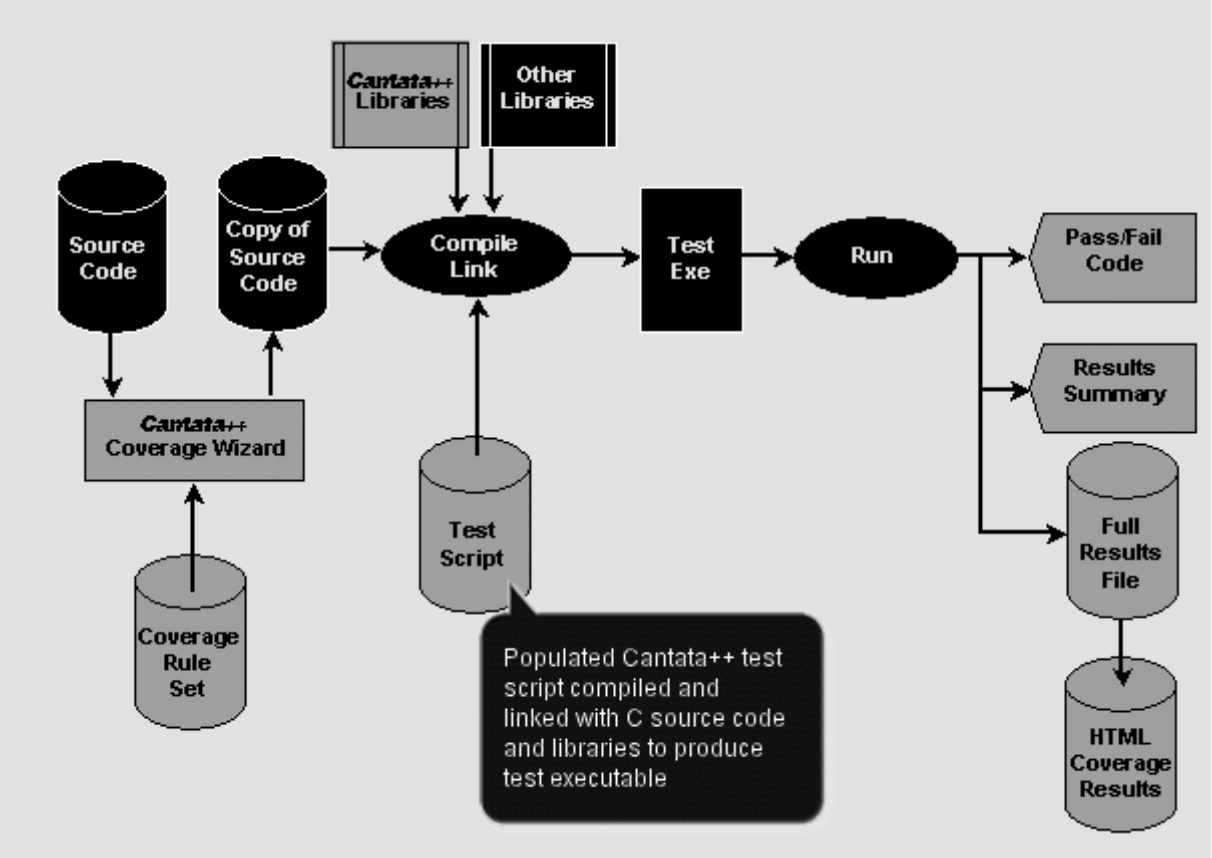
Támogató eszközök:

- Teszt végrehajtó (test harness): Feladata a tesztek futtatása (teszt bemenetek biztosítása, teszt szekvenciák ütemezése). Általában rugalmas script nyelven készül. A teszt robot végzi az ismételt teszteket.
- Teszt értékelő (test oracle): Feladata a teszt kritériumok vizsgálata (teljesítés vagy kudarc megállapítása), egybeépíthető a teszt végrehajtóval is.
- Felműszerező (instrumentation): A teszthez szükséges állapotok azonosítása, a fedettségi statisztikák gyűjtése és a beavatkozás kapcsolódási pontokat igényel, amiket a tesztelt programban kell elhelyezni.
- Teszt csonk (test stub): Feladata a hívott modulok helyettesítése (a teszteléshez szükséges minimális funkció biztosítása, illetve itt is a teszt adatok használata).

Professzionális eszközök:

- Cantata:
  - Teszt scriptek készítése minták segítségével (Test Script Generator): a tesztelendő modul futtatása különféle adatokkal.
  - Felműszerezés, majd a tesztelendő kód ”egybeszerkesztése” a teszt végrehajtóval, értékelővel és fedettség analizátorral: könyvtári függvények illetve makrók használata (pl. CHECK direktívák beépített adattípusokhoz, saját típushoz automatikus ellenőrző generálás, memória blokk hasonlító).
  - Fedettségi analízis automatikus számítása (kritériumok választhatók), kód színezés és statisztika megjelenítése.
  - Statikus analízis is támogatott (program mértékek meghatározása).
  - Teszt konfigurációk kezelése.
- IBM Rational eszközök:
  - QualityArchitect: Teszt végrehajtó és teszt csonk generálás UML diagramokból, teszt adatok felvétele (táblázatos vagy véletlen).
  - Robot: Teszt szekvenciák rögzítése scriptek formájában, ezek módosítása, automatikus lejátszása.
  - Purify: Memóriakezelési problémák kiderítése monitorozással és védelmi funkciókkal.

– Quantify: Teljesítményadatok gyűjtése a tesztelés közben (szűk keresztmetszet azonosítása).
– PureCoverage: Teszt fedettség követése, mértékszámok felvétele.
– Test RealTime: Keresztfejlesztés támogatása, valósidejű követelmények monitorozása.
– Test Manager: Teszt esetek kezelése, tesztelés és eredmények naplózása.



4. ábra. Fedettségi jellemzők kinyerése a tesztelés során a Cantata eszközzel

11. Szoftver integráció

Cél: Fokozatos modul integráció annak érdekében, hogy a szoftver konstrukcióban leírtak teljesüljenek, valamint hogy az együttműködést és az interfészeket még a rendszer verifikáció előtt részleteiben vizsgálni lehessen.
Bemenetek:

- Szoftver konstrukció specifikáció.
- Szoftver integrációs teszt terv.

Kimenetek:

- Szoftver integráció tesztjelentés.
  Ennek dokumentálnia kell a következőket:

  - Tesztesetek és tesztadatok, valamint a teszt kritériumok.
  - Tesztkörnyezet, eszközök és programok.
  - Szoftver konfiguráció az adott teszt elvégzéséhez.

Módszerek: Ld. a modul tesztelés fejezetben ismertetett módszereket; ezek közül itt használandó a részenként összerakott rendszer funkcióinak tesztelésére:

- Funkcionális tesztelés.
- Interfész tesztelés.

11.1. Tesztelési stratégiák

Elsődleges szempont a megfelelő *tesztelési stratégiák és folyamatok* megválasztása, hogy az összerakással egyidejűleg az integrációs tesztelés is elvégezhető legyen.

- Izolációs tesztelés: Az egyes modulokat egymástól elkülönítve teszteljük. Ez a modul tesztelés fázisban megtörténik, ebben a fázisban esetleg csak a kereskedelmi (Commercial Off-the-shelf, COTS) komponensek tesztelésére kerül sor.
- ”Big bang” tesztelés:
  - Teljesen összerakott szoftver rendszer integrációs tesztelése; hiba esetén a hibakeresés és javítás nehéz.
- Általános inkrementális tesztelés:
  - (Nem hierarchikus modul-struktúra esetén) a modulok közötti együttműködés vizsgálata a modulokat egymás után (inkrementálisan) összerakra és tesztelve.
- Felülről lefelé (top-down) tesztelés:
  - Modulok a már tesztelt hívó modulokból kerülnek tesztelésre.
  - Csonkok szükségesek (később ezek helyébe tesztelendő modulok lépnek).
  - Erősen követelmény-orientált (”fentről” tesztelünk, verifikációhoz illik).
  - Modul módosítása az alatta lévők tesztelését módosítja.
- Alulról felfelé (bottom-up) tesztelés:
  - Tesztelendő modul a már teszteltet használja.
  - Tesztvégrehajtó szükséges (később ennek helyébe a tesztelt modul lép).
  - Integrációval (ami tipikusan alulról felfelé történik) együtt végezhető.
  - Modul módosítása a felette lévők tesztjére hatással van.
- Futtatórendszer (backbone) integrációja: Általában nehéz a futtatórendszer (operációs rendszer, köztesrétegek, kernel modulok stb.) helyettesítéséhez teszt csonkokat írni, ezért ezeket nem helyettesítik a felülről lefelé történő tesztelés során. A tesztelés menete:

  - A futtatórendszer alulról felfelé történő tesztelése:
    * Futtatórendszer komponensek izolációs tesztje.
    * Futtatórendszer komponenseinek összeállítása és ”big bang” tesztelése teszt meghajtóval.
    * Futtatórendszer tesztje az alkalmazás modul hierarchia legalsó szintjével (mint teszt meghajtóval).
  - Ezzel párhuzamosan az alkalmazás modulok felülről lefelé történő tesztje a modul hierarchia legalsó szintjéig.
  - Futtatórendszer és alkalmazás modulok integrációja, felülről lefelé történő tesztelés befejezése.

- Szál (thread) tesztelés:

  - A bemeneti eseményeket feldolgozó processzek illetve szálak tesztelése; először egyenként, majd komplex forgatókönyvek alapján is.

Eszközök:

- Wrapper (csomagoló) kód a hívó és a hívott modul közé:

  - Előzetes (before) wrapper: Paraméterek és hívási szekvencia ellenőrzése és módosítása.
  - Helyettesítő (replace) wrapper: A teljes hívott modul lecserélhető.
  - Utólagos (after) wrapper: Visszatérési érték ellenőrzése és módosítása.

12. Szoftver-hardver integráció és rendszertesztelés

Cél: Annak bizonyítása, hogy a szoftver és hardver együttműködése megfelelő ahhoz, hogy előirányzott funkcióikat teljesíteni tudják.
Bemenetek:

- Szoftver architektúra specifikáció.
- Szoftver-hardver integrációs teszt terv.

Kimenetek:

- Szoftver-hardver integráció teszt jelentés.
  A dokumentáltságra hasonlóak érvényesek, mint a szoftver integráció esetén, de itt fokozottan figyelni kell a következőkre:

  - A hardver konfiguráció meghatározása.
  - A telepítés és az integráció feladatainak elkülönítése.
  - A hardver módosítás hatásainak elemzése és az (újból) elvégzendő tesztek meghatározása.

Módszerek: Itt a valós hardverre telepített rendszeren kell a funkciók tesztelését elvégezni.

- Funkcionális tesztelés: Ld. a modul tesztelés fejezetben ismertetett módszerek között.
- Teljesítmény tesztelés.
- Megbízhatóság tesztelés.

12.1. Teljesítmény tesztelés

A teljesítmény tesztelés feladata a specifikációban található teljesítmény és válaszidő jellegű követelmények teljesítésének vizsgálata (ilyen szempontból készített tesztekkel).

- Lavina- és stressz tesztelés: Különösen nagy terheléssel történő tesztelés; ha ez sikerül, akkor a normál terhelést is elviseli a rendszer. A terhelést a funkciótól függően a bejövő kérések számának, a lekérdezés gyakoriságának, a kezelt adatok számának (adatbázis méretének), a változások gyakoriságának szélső értékeivel lehet biztosítani. Közben a belső pufferek, vermek stb. méretének lekérdezésével a terhelésváltozás hatása ellenőrizhető.
- Válaszidő és memória kikötések: A tesztelés közben mérésekkel vizsgálni a jelzett kikötéseket.

A hibakeresésre, a teljesítmény- és időmérésre valamint a terhelésváltozás hatásainak vizsgálatára leginkább a különféle *monitorozó eszközök* használhatók.

12.1.1. Alapműveletek a monitorozáshoz

Három alapfeladat vár megvalósításra:

- Információ hozzáférés: *Felműszerezés*.
- Információ szűrés: *Triggerelés* (a megfigyelni kívánt feltételek teljesülésére várni).
- Információ tárolás: *Regisztrálás* (a megfigyelni kívánt állapot feljegyzése).

Nehézségek, problémák a monitorozás során:

- Beavatkozás: Ha a rendszer erőforrásait használjuk (ld. triggerelés, regisztrálás), akkor megváltoztatjuk a rendszer viselkedését (eltérő időviszonyok miatt eltérő ütemezés, eltérő eseménysorrend miatt akár holtpont is kialakulhat illetve elfedődhet). Megoldás:
  - Hardver monitorozás (legkevésbé avatkozik be).
  - Zavarás (perturbáció) korrekciója: pl. időzítések korrigálása.
  - Monitorozó kód bennhagyása a végleges rendszerben.
- Szemantikai hézag: A regisztrált információból a hasznos információ származtatása (pl. buszjelek alapján következtetni a szemaforműveletekre és az ebből adódó holtpontra.)
- Globális állapot (elosztott rendszerekben): Lokálisan gyűjtött információkból kell meghatározni.
  - Központi monitor használata dedikált kommunikációs csatornákkal.
  - Elosztott monitorok szinkronizálása (globális óra, időbélyegek, esemény-sorrendezés).

12.1.2. Szoftver alapú monitorozás

- Felműszerezés: Trigger utasítások beszúrása.
  - Kézi (csak az ”érdekes” eseményekre) vagy automatikus (esemény-osztályokra, pl. szemafor műveletek) lehet.
  - Kernel (ütemezés is megfigyelhető, alkalmazás-független) vagy az alkalmazás a felműszerezés célja.
  - Bennmaradó (ha nem okoz függőséget) vagy leválasztható.
- Triggerelés: Trigger utasítások jeleznek.
  - Alkalmazás felé vagy kernel felé.
- Regisztrálás: Események feljegyzése.
  - Alkalmazás (közös memóriában) vagy monitor processz végzi.
- Példa: Profilerek (pl. gprof, OProfile), monitorozás aspektus-orientált programozással.

12.1.3. Hardver alapú monitorozás

- Felműszerezés: Csatlakozás busz vezetékekhez.
- Triggerelés: Hardver egység jelmintákat figyel (mintaillesztés).
- Regisztrálás: Busz monitor (cirkuláris puffer), illetve ezek szinkronizálása elosztott rendszerekben. A regisztrált jeleket értelmezni kell (az alacsonyszintű információból magasabb szintűt kinyerni, pl. buszjelekből szemafor műveletek).

12.1.4. Hibrid monitorozás

- Felműszerezés: Trigger utasítások beszúrása (így rugalmas lehet).
- Triggerelés: Trigger utasítások jeleznek.
- Regisztrálás: Hardver monitor (így csökkenthető a beavatkozás).
- Példa: Monitor koprocesszor vagy elosztott monitor.

12.1.5. Hibakeresés technikái

- Valósidejű megjelenítés (”oszcilloszkóp”):
  - Kevés információ áttekinthető, felfüggeszteni nem lehet.
- Valósidejű hibakeresés (”tároló oszcilloszkóp”):
  - Monitorba kell építeni a hibadetektálást (pl. time-out).
- Interaktív hibakeresés, ha megállítható a rendszer (”debugger”):

  - Valósidejű megjelenítés és beavatkozás.

- Determinisztikus visszajátszás (”napló megjelenítés”):
  - Off-line visszajátszás és elemzés (részletgazdag naplózás szükséges).
- Dinamikus szimuláció (”szimulátor”):
  - Újrafuttatás (más teszt adatokkal).

12.1.6. A monitorozott adatok grafikus megjelenítésének eszközei

- (Színezett) processz együttműködési gráf: életvonalak színezése a futó, várakozó, futásra kész állapot alapján.
- Processz státusz gráf: szülő-gyermek kapcsolatok és színezés is.
- Szemafor allokációs mátrix: szemafor - processz összerendelés állapotokkal.
- Üzenet-szekvencia diagram.

12.2. Megbízhatósági és biztonsági tesztelés

A megbízhatósági és biztonsági tesztelés feladata a specifikációban található szolgáltatásbiztonság jellegű követelmények teljesítésének vizsgálata. Ennek legelterjedtebb módja a hibainjektálás. Általában a futtató rendszer, perifériák és illesztő áramkörök állandósult illetve tranziens hibáinak hatását vizsgálják. A hibainjektálás módja:

- Véletlen vagy reprezentatív helyekre történő.
- Valós, tipikus (benchmark) vagy szintetikus terhelés mellett.
- Teljes vagy fontossági jellegű (importance sampling) monitorozás mellett.

A hibainjektálással történő kísérleti értékelés szerepe az egyes életciklus fázisokban:

- *Tervezési fázisok*: Hibaszimuláció (hibadetektálás hibafedése, átkonfigurálás ellenőrzése).
  - Áramköri szint: Valódi fizikai (paraméter)változások részletes vizsgálata (pl. analóg illesztő áramkörök).
  - Logikai szint: Bináris vagy kiterjesztett értékkészletű elemek (beragadási hibák, invertálás).
  - Funkcionális szint: Komplett részegységek illetve rendszerek modellezése (pl. regiszter transzfer szint, processz szint).
- *Integrációs és prototípus fázis*: Hibainjektálás és monitorozás kontrollált terhelés mellett.
  - Hardver alapú: Jelvezetékek (chip lábak) befolyásolása, pl. tápfeszültség tüske szimulálás. Gyors, nagy felbontású, de drága.
  - Szoftver alapú: Rendszerváltozók (pl. processzor regiszterek, memóriabitek) megváltoztatása. Rugalmas, hibamodellje hiányos (csak a hibaállapot jelenik meg). Állandósult hibák többszörös injektálással reprezentálhatók. Kérdéses, hogy valós hibaok következményét injektáljuk-e?

  - Radioaktív alapú: A mérendő rendszer nehézion besugárzása. Általános a hibamodell (tranziens hibák), de kevésbé kontrollálható injektálás. Kérdéses a hibafedés kiszámítása; általában csak az aktív és a detektált hibák aránya mérhető.

- *Működési fázis*: Adatgyűjtés a valós rendszer hibáiról (detektált hibák, statisztikai vizsgálatokkal). Az egyes lépések:

  - Adatfeldolgozás (hibatípusok, korrelált hibák, okok-következmények azonosítása).
  - Modell identifikáció (általában Markov-láncok), és komponens szintű meghibásodási gyakoriság kinyerése.
  - Modell megoldás: Rendszerszintű hatások (MTBF, megbízhatóság időfüggvény) számítása, előrejelzés.

13. Szoftver validáció

Cél: Az integrált rendszer vizsgálata annak megállapítására, hogy az a követelményeknek eleget tesz (beleértve a funkcionális és a nem-funkcionális követelményeket is).
Bemenetek:

- Minden hardver és szoftver dokumentáció, különösen a
- Szoftver követelmény-specifikáció,
- Szoftver validációs terv.

Kimenetek:

- Szoftver validációs jelentés.

  - Dokumentálja a hardver és szoftver konfigurációt, az alkalmazott berendezéseket, a szimulációs modelleket, a végrehajtott konfigurálási esetleg javítási tevékenységeket.
  - Rögzíti a végzett teszteket (ismételhető formában), valamint azok eredményét.
  - Kijelenti, teljesültek-e a validáció kritériumai, az alkalmazó elvárásai. A nem teljesült követelményeket külön fejezetben rögzíti.

Követelmények:

- A validáció támogatására a szimuláció és a modellezés is alkalmazható.
  A szoftvert az üzem alatt terhelni kell a következő jelek szimulációjával:

  - rendeltetésszerű jelek,
  - ritka szituációk (pl. előrehozott illetve késleltetett jelek),
  - beavatkozást követelő nemkívánatos jelek és feltételek.

- A validációs stratégiában el kell különíteni a következő típusokat:

  - kézi vagy automatizált módszer,
  - statikus vagy dinamikus elemzés,

  - analitikus vagy statisztikus módszer.

- A validációhoz használt mérőberendezéseket kalibrálni kell, ezeknek az adott célra való alkalmasságát bizonyítani kell.

Módszerek (leírásukat ld. az előző fejezetekben):

- Teljesítmény tesztelés.
- Funkcionális (fekete doboz) tesztelés.
- Valószínűségi tesztelés.
- Modellezés a környezet szimulációjához (adatáramlási diagramok, állapotátmeneti diagramok, formális módszerek, teljesítmény modellezés, időbeli Petri-hálók, prototípus animáció, struktogramok).

14. Szoftver értékelés

Cél: Annak *minősítése*, hogy az életciklus során végrehajtott eljárások és azok következményei olyanok, hogy a szoftver megfelel a kijelölt biztonság-integritási szintnek és alkalmas a kijelölt feladatra.
Bemenetek:

- Az összes hardver és szoftver dokumentáció, különösen a rendszerbiztonsági követelmény-specifikáció.

Kimenet:

- Szoftverértékelési jelentés.
  - Részletezi az értékelés folyamatát és eredményeit. Értékelendő a biztonság-integritási szint, a személyzet és felelősség, az életciklus és dokumentáció valamint az életciklus során választott és végrehajtott módszerek.
  - Az értékelés eredményeként rögzíti a következők valamelyikét:
    * biztonságintegritási szintnek való megfelelőség,
    * nem-megfelelőség (csak ezt a tényt kell rögzítenie, nem tehet javaslatot a műszaki megoldást illetően),
    * további verifikációs és validációs tevékenység előírása.

Követelmények:

- A 0 biztonságintegritási szint esetén az értékelőnek csak meg kell erősítenie, hogy ez a megfelelő szint.
- Már értékelt szoftver esetén az értékelést nem kell újra végrehajtani, csak a szintet kell megerősíteni.
- A független értékelő feladata eldönteni, megfelelő módszerek kerültek-e végrehajtásra. A tesztelés során jelen lehet.

Módszerek:

- Ellenőrzőlisták (pl. a szabvány előírásai egy ellenőrzőlista elemeinek tekinthetők).
- Statikus és dinamikus elemzés áttekintése az életciklus során.
- Talált hibák és hiányosságok hatásának elemzése:
  - Logikai: Hibafa elemzés, szoftverhiba hatáselemzés, közös eredetű meghibásodások elemzése.
  - Sztochasztikus: Markov-modellek, megbízhatósági blokkdiagramok.
  - Kísérleti: Üzembehelyezés előtti próbák előírása.

15. Szoftver karbantartás

Cél: Biztosítani, hogy a szoftver a rajta végzett változtatások után is az előírt módon működik, megtartva a biztonság-integritási szintet.
Bemenet:

- Szoftver karbantartási terv.
  - Karbantartás ütemezése (mi indítja, időközök).
  - Jelentések, naplók, feljegyzések, engedélyek kezelése és ellenőrzése.
  - A karbantartáshoz kapcsolódó verifikáció, validáció és értékelés.
  - A megváltoztatott szoftver engedélyezési folyamatának leírása.

Kimenet:

- Szoftver változtatási feljegyzések (napló).
  Minden *tevékenységhez* rögzíteni kell:
  - Módosításra vonatkozó igény (problémabejelentés), ennek elfogadása vagy elutasítása.
  - A módosítás teljes rendszerre való hatásának elemzése (hardver, szoftver, környezet, üzemeltetők).
  - A módosítás specifikációja.
  - A módosítás ismételt verifikációja, validációja és értékelése (csak a megváltozott modul, minden érintett modul vagy a teljes rendszer).
- Szoftver karbantartási feljegyzések.
  Minden *szoftverelemhez* (modulhoz) rögzíteni kell még a használatbavétel előtt (pl. a verziókövető rendszerben):
  - Hivatkozási rendszer a változtatási feljegyzésekre.
  - Hivatkozási rendszer a változtatások következményeire.
  - Tesztesetek az ismételt verifikációhoz és validációhoz.
  - Szoftver konfiguráció történet.

Módszerek:

- Hatáselemzés (M): Hiba hatása modul- illetve rendszerszinten, hibadetektálás és -kezelés, bennmaradó kritikus ügyek.
- Adatrögzítés és elemzés (M): Döntések, határpontok, problémák és megoldások rögzítése. Példák a hibabejelentő rendszerek funkcióira:
  - Probléma bevitele és jellemzése (pl. súlyosság, gyakoriság, típus, prioritás).
  - Felelős hozzárendelése, jogosultságok kiosztása.
  - Probléma életciklus követése (pl. nyitva, elemzés alatt, további információra vár, javítva, tesztelve, lezárva), értesítések küldése.
  - Probléma továbbítás az egyes szintek között (pl. modul, alkalmazás, rendszer).
  - Megelőző jellegű tevékenységek.
  - Vezetői lekérdezés az egyes kategóriákról (pl. függő problémák, felelősök terhelése).

Példák:

- GNATS problémajelentő és kezelő rendszer (GNU, webes felület). Bejelentkezés; műveletek olvasói (csak problémabejelentés és -követés), szerkesztői (státusz módosítás is) vagy adminisztrátori jogokkal.
- Rational ClearQuest. Változáskezelés indítható a TeamTest, Purify, PureCoverage, Quantify, Visual Test programokból. *Munkafolyamat sémák* összeállítása lehetséges: változáskérés rekord elemei, benyújtáshoz és módosításhoz használható űrlapok, státusz definíciók, akció definíciók (az egyes állapotváltozások között elvégzendő műveletek), szkriptek az űrlap mezők és akciók finomításához (pl. automatikusan beállítják a változáskezelés határidejét). Ezek a sémák a verziókövetés hatálya alatt vannak, tehát a módosítások követhetők és az adminisztrátorok kölcsönös kizárása is biztosított.