Szolgáltatásbiztonságra tervezés laboratórium (VIMIM236)

Hibadiagnosztika elosztott rendszerekben

Mérési segédlet

Készítette: dr. Bartha Tamás

Utolsó módosítás: 2010. október 25.

Verzió: 1.0

Budapesti Műszaki és Gazdaságtudományi Egyetem
Méréstechnika és Információs Rendszerek Tanszék

1 Bevezető

A labor mérései során egy bonyolultabb többrétegű alkalmazást kiszolgáló összetett infrastruktúra szolgáltatásbiztonságát vizsgáljuk különböző technikák segítségével. A rendszer lehetséges felépítési alternatíváit modellezéssel és mérésekkel elemezzük, az így azonosított hibamódok és szűk keresztmetszetek kiküszöbölésére pedig különböző hibatűrést és skálázhatóságot segítő technológiákat próbálunk ki a gyakorlatban. A mérések felépítése a következő:

1. Konfigurációs leírók modell alapú fejlesztése
2. Terheléselosztó fürtök és teljesítménytesztelés
3. Feladatátvételi fürtök
4. Megbízhatóság modellezés
5. Hibadiagnosztika
6. Robosztusság vizsgálata hibainjektálással
7. Szolgáltatásbiztonsági benchmark

A jelen mérés során a *hibadiagnosztikával*, azaz a hibák helyének a teszteredmények alapján történő pontos behatárolásával foglalkozunk. Megismerjük a *rendszerszintű, központosított hibadiagnosztika* módszerét, amely esetén egy elosztott rendszer feladatvégrehajtó egységei egymáson hajtanak végre teszteket, majd az összegyűjtött teszteredményeket egy központi kiértékelő algoritmus dolgozza fel és azonosítja a meghibásodott egységeket.

2 A hibadiagnosztika feladata

A korszerű számítástechnikai eszközökben használt processzorok és kiegészítő áramkörök teljesítménye folyamatosan nő, mégis az egyre magasabb teljesítmény igényeket az egyetlen CPU-n alapuló rendszerek sokszor nem tudják kielégíteni. Ezért napjainkban ismét „reneszánszukat élik" az ún. *architekturális teljesítménynövelő módszerek*. Ezen módszerek leggyakrabban alkalmazott formája a *párhuzamos feladatvégrehajtás* (parallelizáció). A párhuzamos architektúrák az integrált áramköri szinttől (több processzormag használata egyetlen CPU lapkán) a különféle terheléselosztó „szerver farmokon " és többrétegű kiszolgálókon át egészen a Grid, Cloud és erősen párhuzamos (massively parallel) számítási rendszerekig terjednek

A megbízható működés eléréséhez elosztott rendszerekben nem elég precízen kialakított és kimerítően tesztelt alkatrészeket felhasználni, mert a párhuzamos feldolgozó egységek használata a meghibásodás valószínűségét is a többszörösére növeli. Ezért a megfelelő rendelkezésre állás eléréséhez az elosztott rendszereknek kezelniük kell a működés közben fellépő hibákat. Ehhez a hibákat időben fel kell fedezniük (hibadetektálás) és azonosítaniuk (diagnosztika), valamint azok hatását el kell tüntetniük anélkül, hogy a felhasználó érzékelhetné.

A hibákból való felépüléshez mindig szükség van valamilyen rendszerbeli redundanciára. Az elosztott rendszerekben ez a redundancia alapvető tervezési adottság. A működés során e rendszerek ritkán érik el a teljes kihasználtságot (amikor minden feladatvégrehajtó egység teljes terheléssel dolgozik). Azaz szinte mindig van tartalék kapacitás. Ezt a tartalékot használhatjuk fel a hibákból való felépülésre úgy, hogy a továbbra is működőképes processzorok átveszik a meghibásodott egységek feladatait.

Mivel a hibák kezelése a feldolgozó egységek szintjén történik, célszerű a tesztelési és diagnosztikai feladatokat is ezen a szinten megoldani. Ezt a megközelítést *rendszerszintű hibadiagnosztikának* nevezik. A processzorok tesztelését nem egy külső egység, hanem maguk a processzorok végzik el. Minden processzor teszteli az elsődleges szomszédait (azokat az egységeket, akik vele közvetlen tesztkapcsolatban vannak), és a kapott teszteredményeket elküldi egy kiértékelő eszköznek. Az összegyűjtött teszteredmények kiértékelésének célja maga a diagnosztika, tehát a tesztelési információ alapján a hibás egységek azonosítása.

Összefoglalva: a hibák hatásának eltüntetésére alkalmazható ún. *on-line hibajavítás* módszere jellemzően a következő lépésekből áll:

1. *Állapotmentés*. A feldolgozó egységek (processzorok) állapota periodikusan mentésre kerül.
2. *Tesztelés*. A hibák felderítése céljából az egységek valamilyen ütemezéssel teszteket hajtanak végre egymáson.
3. *Diagnosztika*. Az összegyűjtött teszteredmény-halmaz (melyet *szindrómának* is neveznek) elemzésével a hibák pontos helyét megállapítja a kiértékelő eszköz.
4. *Hiba izoláció*. A diagnosztikai információ alapján a meghibásodott egységek kizárásra kerülnek a további feldolgozási folyamatból.
5. *Rekonfiguráció*. A feladatok újraosztásra kerülnek a működőképes egységek között, a meghibásodott egységek feladatait a hibátlan vagy az újonnan bekapcsolt tartalék egységek veszik át.
6. *Felépülés*. A rendszer normál (de esetleg degradált) működésének visszaállítása egy *korábbi hibátlan állapotot* tartalmazó állapotmentés (visszalépés), vagy egy megfelelő algoritmus segítségével meghatározott *új hibátlan rendszerállapot* (előrelépés) segítségével.

2.1 Tesztelési és diagnosztikai alapfogalmak

Elsőként a rendszerszintű hibadiagnosztikával kapcsolatos legfontosabb fogalmakat ismertetjük.

A magyar „hiba” szóval több különböző fogalmat is jelölünk. Ezek definíciója csak az angol terminológiában válik el élesen, ezért a fogalmak bemutatásakor mindkét megnevezést megadjuk:

- **Hibajelenség** (Failure): a rendszer vagy szolgáltatás felhasználója által észlelt, a funkcionális vagy minőségi specifikációtól eltérő működés.
- **Hibaállapot** (Error): a rendszernek a normálistól eltérő, megváltozott állapota, amelynek fellépte kivált(hat)ja a hibajelenség létrejöttét. Amíg egy hibaállapotot nem aktivizálódott, azaz nem hozott létre hibajelenséget, addig látensnek, azaz lappangónak nevezzük.
- **Hibaok** (Fault): A hibaállapotot kiváltó fizikai sérülés, meghibásodás.

Két példa a fenti fogalmak jobb megértéséhez:

1. *Hibaok* egy programozói hiba, aminek következménye egy látens *hibaállapot* a programban (hibás utasítás vagy adat). Ez a lappangó hibaállapot a megfelelő körülmények között aktivizálódik. Amikor az aktív hibaállapot a program kimenetén megjelenik (pl. hibás számítási végeredmény formájában), akkor létrejön a *hibajelenség.*
2. *Hibaok* egy rövidzár vagy szakadás egy integrált áramkör belsejében. A létrejövő hibás logikai érték vagy módosult kapu leíró függvény a *hibaállapot*. Ha aktivizálódik és az integrált áramkör funkcióját a felhasználó által észrevehetően befolyásolja (pl. a számítógép lefagy memória-hozzáfordulás hiba miatt), akkor *hibajelenség* alakul ki.

Általában a számítógéprendszerek hierarchikusan szervezettek, és így szintekre bonthatók. Ilyenkor a hibaok $\rightarrow$ hibaállapot $\rightarrow$ hibajelenség hatásmechanizmust a hierarchia szinteken belül is értelmezhetjük. Egy adott szinten létrejövő hibajelenség a következő szinten hibaokként jelenik meg. Ott hibaállapotot majd a szint határáig terjedve hibajelenséget hoz létre, ami egy szinttel feljebb lépve ismét hibaokként jelentkezik, és így tovább.

A következőkben ismertetjük a teszteléssel kapcsolatos alapfogalmakat (a definíciók a rendszer hierarchia egy adott szintjére értelmezettek):

- **Hiba, hibaminta**. A tesztelés, teszttervezés első lépése a rendszer *hibamodelljének* felállítása. Ennek során meg kell határozni a rendszer különféle meghibásodási módjait, fel kell térképezni az összes komponensre a lehetséges hibaokokat és hibaállapotokat. Ha a hibamodell elkészült, akkor meg kell vizsgálni minden hibaállapot esetén, hogy az adott komponens a bemeneteire adott tesztvektorokkal hogyan *vezérelhető* olyan állapotba, amely kívülről is *megfigyelhetően* eltér a hibás és hibátlan esetben.

Tegyük fel, hogy a rendszer $p = |\Phi|$ darab hibát tartalmaz. Az egyes hibák jelölése φ_i. A hibakészlet ezen hibák halmaza: $\Phi = \{\varphi_1, \ldots, \varphi_p\}$. A hibakészlet egy j. részhalmazát *hibamintának* nevezzük, jelölésére egy bináris értékekből álló vektor használunk $F_j = (f_1, \ldots, f_p)$ ahol f_i az 1 értéket veszi fel, ha a φ_i hiba jelen van a rendszerben, 0 értéket, ha nincs. Az összes lehetséges hibamintából alkotott halmazt F-fel jelöljük: $F = \{F_0, \ldots, F_x\}$.

- **Teszt, tesztminta**. Általánosságban egy *teszt* a vizsgált berendezés bemeneteire adott tesztvektorokat és a berendezés egy vagy több kimenetén a tesztvektorokra adott válaszknak a vizsgálatát jelenti. A tesztelt egység jellegétől függően az általános definíciónak megfelelő a teszt sokféle formát ölthet: megvalósítható szoftverben vagy hardverben, és végezheti a tesztet beépített tesztelő komponens vagy külső tesztelő berendezés.

  A tesztekre alkalmazott jelölésrendszer a hibáknál bemutatotthoz hasonló. Egy teszt jelölése τ_i. A rendszeren értelmezett összes teszt száma $q = |\Theta|$, a tesztkészlet ezek halmaza: $\Theta = \{\tau_1, \ldots, \tau_q\}$. A j. *tesztminta* a teszteredményekből alkotott vektor: $Tj = \{t_1, \ldots, t_q\}$, ahol t_i az 1 értéket veszi fel, ha a τ_i teszt hibát mutatott ki, és 0 értéket ha hibátlan volt. A teszteredmények összességét más néven *szindrómának* is nevezzük. Az összes lehetséges tesztminta halmaza: $T = \{T_0, \ldots, T_y\}$.

- **Eseménytér**. Az eseménytér a hibaminta ↔ tesztminta kapcsolatot leíró táblázat. Megadja, hogy egy adott hibaminta hatására a teszteredmények milyen értéket vehetnek fel, azaz milyen tesztminták jöhetnek létre. Egy hibamintához több tesztminta is tartozhat, ennek oka a diagnosztikai bizonytalanságot okozó két hatás: a *nem teljes hibafedés* és a *tesztérvénytelenítés*.

- **Teljes teszt**. Egy tesztet egy adott hibára nézve *teljesnek* (100 százalékos hibafedésűnek) nevezünk, ha mindig hibás eredményt ad, amikor az adott hiba jelen van a rendszerben és mindig hibátlanul fut le, ha a hiba nincs jelen.

- **Hibafedés**. Ha egy teszt *nem teljes*, akkor az adott hibát nem minden esetben, hanem csak az esetek egy részében, bizonyos valószínűséggel mutatja ki. Ezt a valószínűséget nevezik az adott teszt *hibafedésének*. A részleges hibafedés oka lehet, hogy az adott teszt nem veszi figyelembe a hibát meghatározó összes hibaállapotot (pl. egy memórateszt, ami nem minden memóriacellát vizsgál, esetleg a cellák 1-be ragadását ki tudja mutatni, de a 0-ba ragadást nem, stb.). Másik lehetséges ok, ha bizonyos hibaállapotok az eszköz korlátozott vezérelhetősége és/vagy megfigyelhetősége miatt nem vizsgálhatók. Ez utóbbi okból a gyakorlatban kellően összetett áramkörökre általában nem is lehet minden hibára nézve teljes tesztet készíteni. A részleges hibafedés ugyanakkor diagnosztikai szempontból nehezen kezelhető.

- **Tesztérvénytelenítés**. A tesztek eredményének használhatóságát jelentősen befolyásolja a tesztet végző eszköz hibaállapota. Rendszerszintű diagnosztika esetén a teszteket maguk a rendszert alkotó processzorok végzik. Ha a tesztelő processzor hibás, akkor az általa végzett teszt érvényét vesztheti. A *tesztérvénytelenítés* jelensége azt írja le, hogyan torzítja az adott típusú tesztelő egység a teszteredményt meghibásodás esetén.

- **Általánosított tesztleképzés**. Az évek során a szakirodalomban különböző megfontolások alapján több eltérő tesztérvénytelenítési modell is napvilágot látott. Az általánosított tesztleképzés a fenti, már létező modelleket, valamint néhány további lehetséges modellt foglal egységes keretbe. Az egyesített modell az 1. táblázat segítségével írható le.

1. táblázat: Az általánosított tesztleképzés

Tesztelő állapota	Tesztelt egység állapota	Teszteredmény
0 (hibátlan)	0 (hibátlan)	0 (hibátlan)
0	1 (hibás)	1 (hibát jelez)
1 (hibás)	0	$c \in \{0, 1, X\}$
1	1	$d \in \{0, 1, X\}$

A táblázatban megadott *c* és *d* betűk szimbolikus konstansok. Ezek helyére rendre a 0, 1 és X értékek valamelyikét helyettesíthetjük. (Az X az ún. *don't care* érték, ami véletlenszerűen akár 0, akár 1 értékű lehet.)

Egy konkrét *c* és *d* érték-hozzárendelés rögzít egy adott tesztérvénytelenítési modellt. Ezzel a módszerrel tehát 3·3, azaz 9 különböző modell adható meg. Az egyes tesztérvénytelenítési modellek jelölése: TI_{cd} (ahol *c* és *d* helyére az aktuális modellnek megfelelő konstansok kerülnek). Felhívjuk a figyelmet a táblázat első két sorára: ezek azt fejezik ki, hogy a hibátlan tesztelő *mindig megfelelően* minősíti a tesztelt egységet. Tehát az általánosított tesztleképzés *teljes tesztekkel* dolgozik!

A jobb megértés kedvéért nézzünk meg két, a szakirodalomban elterjedt tesztérvénytelenítési modellt! Történetileg az első a *Preparata* vagy gyakoribb nevén *PMC modell* volt (a rövidítés a modell alkotóinak kezdőbetűit rejti). Mindmáig ez a legtöbbet használt tesztérvénytelenítési séma, lévén a „leginkább pesszimista". Ugyanis a modell szerint a hibás tesztelő bárhogy viselkedhet: mondhatja akár a hibátlan tesztelt egységről azt, hogy hibás; akár a hibás tesztelt egységről azt, hogy hibátlan. Ezt az általánosított tesztleképzés fogalmai szerint a *c* = X és *d* = X hozzárendelés fejezi ki, vagyis a PMC modell jele: TI_{XX}. Ennél a modellnél tehát a hibás tesztelő teszteredménye *közvetlenül nem vehető figyelembe*.

A gyakorlatban a tesztelés sokszor a következő séma szerint történik:

1. a tesztelő elküld a tesztelt egységnek egy adatsorozatot,
2. az adatsorozaton mind a tesztelő, mind a tesztelt egység elvégzi ugyanazt a műveletsort (mely lehetőség szerint a tesztelt egység minél több funkcióját használja,
3. a tesztelt egység visszaküldi a végeredményt, melyet a tesztelő a saját etalonnak tekintett eredményével hasonlít össze.

Ha a két teszteredmény eltér, a teszt hibát jelez. Ilyen esetekre bizonyos megszorításokat tehetünk, melyet az ún. *BGM modell* ír le, aminek tesztleképzése TI_{X1}. Alapgondolata: nagyon kicsi annak a valószínűsége, hogy a tesztelő és a tesztelt egység teljesen azonos módon romlik el. A jó hibafedés érdekében a tesztelő nem egy, hanem kellően sok különböző tesztet végez el, így a két hibás egység hibamódjának különbsége végül felfedezhető. Mindkét egység együttes hibája esetén az összesített teszteredmény ezért lesz 1 (sikertelen).

A BGM és a hozzá hasonló tesztérvénytelenítési modellek a PMC modellhez képest megszorításokat tartalmaznak, amit a rendszerről ismert többlet információ alapján tehetnek meg. Ezen modellek jelentősége abban áll, hogy kevesebb diagnosztikai bizonytalanságot tartalmaznak, ami jelentősen megkönnyíti a szindróma kiértékelését. (Például BGM modell esetén, ha egy tesztelt egységet valamely tesztelő 0 (sikeres) eredménnyel tesztelt, akkor a tesztelt egység *biztosan hibátlan*. Ennek az állításnak az igazolását az 1. táblázat alapján az olvasóra bízzuk.)

- **Tesztelési gráf**. Ez a tesztelő → tesztelt egység kapcsolatokat leíró irányított gráf, melynek csomópontjai a rendszert alkotó processzorok, és benne minden tesztelőtől az általa tesztelt egységekhez egy irányított él húzódik. Mint már említettük, tesztkapcsolat csak közvetlen szomszédok között állhat fenn, így a tesztelési gráfot elsősorban a többprocesszoros rendszer kommunikációs topológiája határozza meg.
- **Detektálhatóság, diagnosztizálhatóság**. Mindkettő a tesztelési gráfokra jellemző mérték. A *detektálhatóság* azt mutatja meg, hogy egy adott gráfban maximum hány hiba fennállása esetén lehetséges egy bekövetkezett új hiba felfedése. A *diagnosztizálhatóság* annak feltételét adja meg, hogy maximum hány hiba fennállása esetén lehetséges a hibás egységek azonosítása. A továbbiakban a rendszert alkotó összes egység számát az *n*, a hibás egységek maximális számát pedig a *t* szimbolikus konstansok jelölik:
    - Egy adott rendszer *t*-hiba detektálható, ha legalább egy teszt sikertelen lefutású, amennyiben a hibás egységek száma ≤ *t*.
    - A rendszer egy lépésben *t*-hiba diagnosztizálható, ha minden tesztet egyszer végrehajtva az összes hibás egység azonosítható, amennyiben a hibás egységek száma ≤ *t*.

Az egy lépésben *t*-hiba diagnosztizálhatóság feltételei PMC tesztérvénytelenítési modell esetén (ne felejtsük el, hogy legfeljebb *t* darab hibás egység lehet a rendszerben):

- a rendszert alkotó egységek száma legalább $n \geq 2t + 1$,
- minden egységet legalább *t* másik egység tesztel,
- nincsenek kölcsönös tesztek.

- A rendszer javítva *t*-hiba diagnosztizálható, ha az egységek cseréje nélkül *legalább egy hibás egység* azonosítható, amennyiben a hibák száma ≤ *t*. Így az éppen azonosított hibás egységet kijavítva legfeljebb *t* lépésben az összes hibát kijavíthatjuk (feltéve, hogy a javítási ciklus alatt nem keletkeznek újabb hibák). A feltételek PMC modell esetén az egy lépésben *t*-hiba diagnosztizálhatóság feltételeihez hasonlóak, de az elvégzendő tesztek száma kevesebb: $n \cdot t$ teszt helyett itt $2n - 2$ teszt is elég (sőt, bizonyos struktúrákban elég $n + 2t - 2$ teszt is).

A diagnosztikai feladat megoldásának kétféle technikai megvalósítása létezik. A gyakorlatban először alkalmazott, és a mérés során általunk is használt *központosított diagnosztika* esetén a rendszerben létezik egy kitüntetett *szindróma kiértékelő* egység. Minden processzor (logikailag) különálló és megbízható kommunikációs összeköttetéssel kapcsolódik ehhez az egységhez. A teszteket a feladatvégrehajtó egységek maguk hajtják végre egymáson, de a teszteredmények kiértékelése már kizárólag a központi egység feladata. Ez az egység irányítja a hibák hatásának elhárítását is.

A mérés során megvalósítandó feladatok két csoportba oszthatók: *determinisztikus* és *valószínűségi* diagnosztikai algoritmusokra.

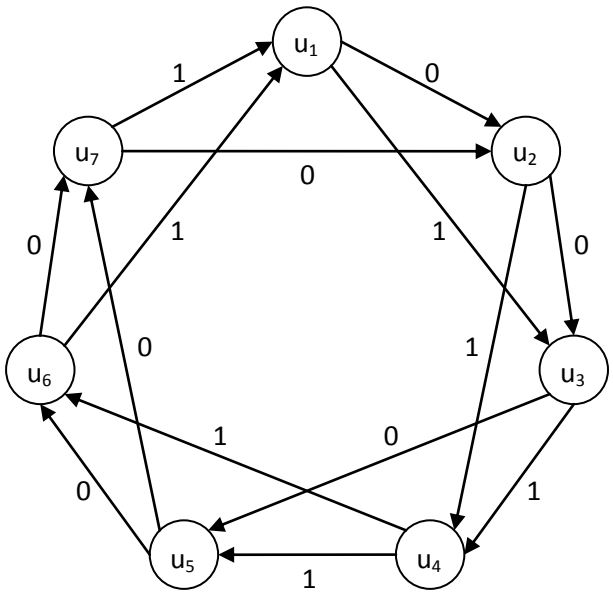
- A *determinisztikus algoritmusok* adott feltételek teljesülése esetén képesek *teljes* és *konzisztens* diagnosztikai képet előállítani. Teljesnek akkor nevezünk egy diagnosztikai eredményt, ha a rendszer minden egysége kapott valamilyen (hibátlan/hibás) minősítést, konzisztensnek pedig akkor, ha minden egység valós és az algoritmus által feltárt hibaállapota megegyezik. A teljes és konzisztens diagnosztika eléréséhez teljesítendő további feltétel(eke)re példa az előbbiekben tárgyalt, a hibás egységek számára adott felső korlát, az ún. *t*-korlát.
- A *valószínűségi algoritmusok* nem garantálnak biztosan teljes és konzisztens megoldást, de nagy valószínűséggel helyes eredményt szolgáltatnak. A tévedés lehetőségért cserébe ezek a módszerek csak a teszteredményekből kinyert információt használják fel, azaz nem igényelnek a rendszerre nézve olyan további előírásokat, mint a *t*-korlát a determinisztikus módszereknél.

  A valószínűségi algoritmusok lehetséges diagnosztikai tévedései három csoportba sorolhatók:

  1. Az *ismeretlen egység* egy olyan processzort jelent, amelynek hibaállapotát az algoritmus nem tudta meghatározni. Ebben az esetben a diagnosztika nem teljes.
  2. *Jóindulatú tévedés* esetén egy hibátlan processzort hibásnak minősítünk. Így csökkentjük ugyan a rendelkezésre álló erőforrások számát, de a biztonság javára tévedünk.
  3. *Rosszindulatú tévedés* esetén egy hibás processzort hibátlannak minősítünk. Ekkor a rendszerből nem távolítjuk el a hibás komponenst, ami hibás adatokkal és hibaterjesztéssel az egész folyamat hibáját okozhatja. Az erőforrások számát nem csökkentjük.

3 Rendszerszintű, központosított hibadiagnosztika

A jelen mérés célja egy rendszerszintű, központosított hibadiagnosztikai algoritmus kidolgozása és kipróbálása. A diagnosztikával kapcsolatos problémákat és megoldásuk lehetséges módját egy egyszerű példán keresztül szemléltetjük. Tekintsünk egy 7 egységből álló elosztott rendszert. Tegyük fel, hogy a rendszer tesztelési elrendezése az 1. ábra látható ún. $D_{1,2}$ topológiájú gráfnak felel meg. Az ábrán a csomópontok a feldolgozó egységeknek, az irányított élek a tesztelő → tesztelt egység kapcsolatoknak felelnek meg. Minden egység PMC tesztérvénytelenítést használ. A teszteredményeket az élekre írt számok jelképezik. (A hibátlanul lefutott teszt eredménye 0, a hibát jelző teszt eredménye 1.)



1. ábra: Egy 7 feldolgozó egységet tartalmazó $D_{1,2}$ topológiájú elosztott rendszer

Ha a szindrómát központosított diagnosztikai algoritmussal elemezzük, akkor első lépésben a tesztek kerülnek végrehajtásra, majd a tesztelő egységek az általuk elvégzett tesztek eredményeit elküldik a központi diagnosztikai komponensnek. Első gondolatunk az lehet, hogy a szindrómát alkotó 0-ák és 1-ek halmazából nem deríthető ki semmi biztos állítás a rendszert alkotó egységek állapotára nézve, hiszen PMC modell esetén a hibás egységek által végzett tesztek eredménye közvetlenül nem használható, és egyelőre nem ismert, mely egységek hibásak. Mégis, pusztán logikai úton is sok hasznos következtetést vonhatunk le a szindrómából az egységek állapotára vonatkozóan:

- A rendszerben van hibás egység, ellenkező esetben a rendszerben az összes teszt hibátlanul futna le. Ez *hibadetektáló* információ.
- Az 1-es egység *biztosan hibás*. Ennek bizonyításához tegyük fel, hogy az egység hibátlan. Ekkor a 2-es egységnek is hibátlannak kell lennie, hiszen őt az 1-es (hibátlannak feltételezett) egység hibátlannak tesztelte. De ebből az következne, hogy a 3-as egység hibátlan (őt a 2-es találta hibátlannak), ugyanakkor hibás is (az 1-es egység tesztje hibát jelzett). Ellentmondásra jutottunk! Tehát az eredeti feltevésünk hibás volt, ebből következően az 1-es egység csak hibás lehet. Ez hibalokalizáló információ.
- A 2-es, 3-as, 5-ös, 6-os és 7-es egységek hibaállapota *azonos*. Ezek az egységek ugyanis körkörösen 0 teszteredményekkel mutatnak egymásra. Ha valamelyikük például hibátlan, akkor az általa tesztelt egység is hibátlan kell legyen. Ha hibás, akkor viszont az az egység is hibás, aki őt tesztelte. Ugyanez igaz a körben levő összes többi egységre is, azaz mindannyian egyszerre lehetnek csak hibásak vagy hibátlanok.

Természetesen az egyik lehetséges diagnosztika az a rendszerről az, hogy minden egység hibás. Mivel a PMC modellben bármely hibás tesztelő egység tetszőleges teszteredményt szolgáltathat a tesztelt egység állapotától függetlenül, bármely szindróma kompatibilis a „totális rendszerhiba” esetével.

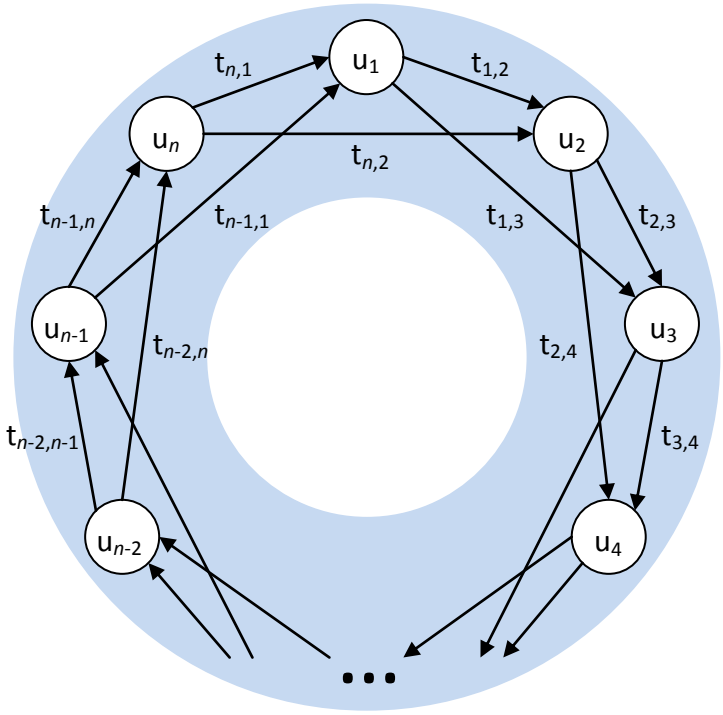
Bár „az összes egység hibás” következtetés kevés gyakorlati jelentőséggel bír, nem hagyható ki a lehetséges következtetések halmazából valamilyen a rendszerről meglevő előzetes plusz ismeret nélkül. Ilyen ismeret lehet például, hogy a rendszerben a hibás egységek száma nem halad meg egy bizonyos korlátot. Természetesen ehhez a rendszert vagy az üzemeltetési körülményeket úgy kell kialakítani, hogy a feltétel megfelelően nagy valószínűséggel teljesüljön. Legyen a hibás egységek maximális száma (az előző pontban már említett diagnosztikai t-korlát) $t = 2$. Ebben az esetben a 2-es, 3-as, 5-ös, 6-os és 7-es egységek biztosan hibátlanok, hiszen az ő hibaállapotuk azonos, és hat hibás egység együttes létezése a t-korlát betartása esetén lehetetlen. Amiből viszont a 4-es egység hibás volta következik.

Ezzel a rendszerben levő összes egységet minősítettük, méghozzá ellentmondás-mentes és a szindrómával kompatibilis módon, azaz egy újabb lehetséges diagnosztikához jutottunk. További vizsgálatokkal még a következők állapíthatók meg:

- Nincs lehetséges diagnosztika, amennyiben csak egy egység lehet hibás.
- Ha a hibás egységek maximális száma 2--5 között mozog, akkor a fenti diagnosztika az egyetlen lehetséges megoldás.
- Ha a *t*-korlátot 6-ra emeljük, akkor a fenti megoldáson kívül még a „4-es egység jó, az összes többi hibás” szituáció is bekerül a lehetséges diagnosztikák halmazába.

3.1 Meyer és Masson algoritmusa

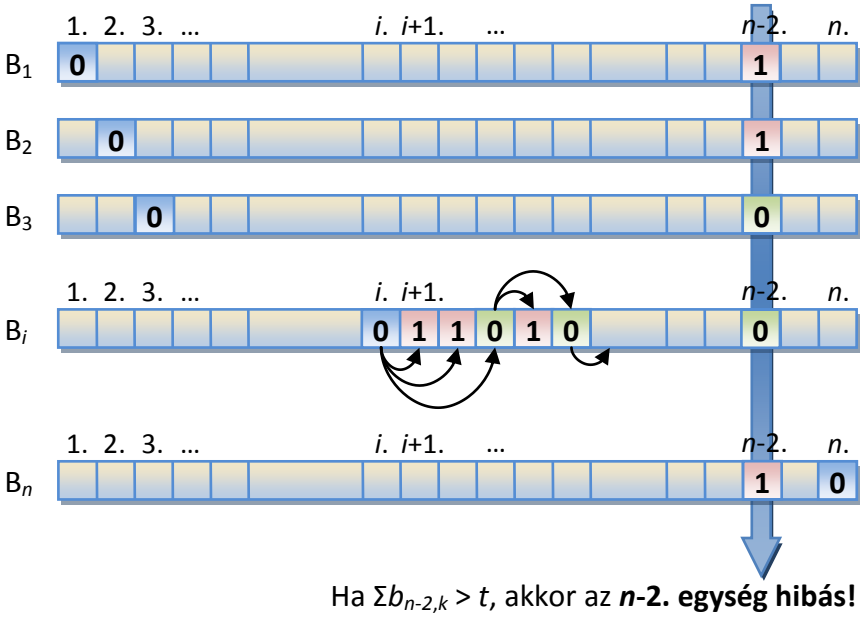
A következőkben ismertetünk egy általános célú, egy lépésben minden egységet minősítő diagnosztikai algoritmust, amit Meyer és Masson dolgozott ki PMC tesztérvénytelenítési modellre és $D_{1,t}$ típusú tesztelési gráfokra. $D_{\sigma,A}$ típusúnak nevezünk egy gráfot, ha a minden csomópontból A darab irányított él indul ki, és az élek a csomópontra rákövetkező minden σ-ik csomópontba mutatnak. Tehát egy $D_{2,3}$ típusú gráfban az 1-es csomópontból három él indul ki, amik rendre a 2-es, 4-es és 6-os csomópontba mutatnak. Egy n darab csomópontot tartalmazó $D_{1,2}$ gráf szerkezetét mutatja be a 2. ábra. (Szintén $D_{1,2}$ típusú topológiát használ az 1. ábrán bemutatott példa rendszer is.) Látható, hogy a $D_{1,t}$ típusú tesztelési gráfok teljesítik a PMC modellre vonatkozó egy lépésben *t*-hiba diagnosztizálhatóság feltételeit.



2. ábra: Egy *n* csomópontú $D_{1,2}$ gráf

A Meyer és Masson algoritmus a következő lépésekből áll:

1. Kiindulásként a tesztelő egységek a tesztelési gráfnak megfelelően végrehajtják a teszteket, majd a központi kiértékelő egység megkapja az összes teszteredményt.
2. A központi egység a teszteredmények alapján készít egy *n* sor és *n* oszlopból álló B táblázatot. Ennek a táblázatnak minden egyes *i*. sora (a továbbiakban ezt B_i jelöli) az *i*. tesztelő egységhez tartozó lokális diagnosztikai képet fogja előállítani.
3. A táblázat inicializálásakor az *i*. sor *i*. oszlopába 0 kerül be. A B_i sor $b_{i,i}$-vel jelölt *i*. elemének nullázása azt jelenti, hogy az *i*. egység a hozzá tartozó lokális diagnosztikai képben hibátlanként szerepel, azaz ebben a sorban egy olyan lokális következtetési lánc indul, amelynek kezdeti feltételezése, hogy az *i*. egység hibátlan.



3. ábra: A Meyer és Masson algoritmus vázlata

4. Ezután a központi egység sorok szerint kitölti a B táblázatot a következő módszerrel:

   4.1. Az aktuális *i*. sorban először az *i*. egység által elvégzett tesztek eredményeiből levonható következtetéseket vizsgálja. Mivel a kiinduló feltevés szerint az *i*. egység hibátlan és a tesztek teljesek, az *i*. egység teszteredményei helyesen minősítik az általa tesztelt egységek állapotát. A központi egység sorrendben megnézi a *k*. szomszédra ($k = 1, 2, \ldots, t$) vonatkozó tesztet (ezek a $D_{1,t}$ típusú tesztelési gráfokban az *i*+1., *i*+2., …, *i*+*t*. egységek).

      4.1.1. Amíg az adott teszteredmény hibát jelez ($t_{i,i+k} = 1$), addig a táblázat B_i sorában a megfelelő *i*+1, *i*+2, … oszlopokba 1-eseket ír be: $b_{i,i+k} = 1$ (3. ábra). Ezek az egységek a kiinduló feltétel alapján hibásnak minősülnek, és az általuk elvégzett tesztek eredményei nem használhatók fel.

      4.1.2. Ha talál egy hibátlan teszteredményt (vonatkozzon ez mondjuk a *j*. szomszédra), akkor az *i*+*j*. tesztelt egységet hibátlannak fogadja el és ennek megfelelően a B_i sor *i*+*j*. oszlopába 0-át ír: $b_{i,i+j} = 0$. A kiinduló feltétel szerint ez az *i*+*j*. egység is hibátlan, azaz a teszteredményei szintén helyesen minősítik az általa tesztelt egységek állapotát.

   4.2. Ezért ettől kezdve az eljárás az immár hibátlannak feltételezett *i*+*j*. egység teszteredményeivel folytatja a táblázat kitöltését. Most már az *i*+*j*. egységnek veszi sorrendben a szomszédokra vonatkozó teszteredményeit és a táblázat kitöltését az *i*+*j*+1, …, *i*+*j*+*k* pozíciókkal folytatja. Ugyanazt teszi, mint az előbb:

      4.2.1. A táblázat *i*+*j*+*k*. elemét 1-el tölti ki: $b_{i,i+j+k} = 1$, amíg a vizsgált *k*. szomszédra vonatkozó $t_{i+j,i+j+k} = 1$ teszteredmény hibás.

      4.2.2. Hibátlan teszteredménynél az *i*+*j*+*k*. pozícióra 0-át ír be: $b_{i,i+j+k} = 0$, és ismét „átkapcsol” az újonnan megtalált hibátlannal feltételezett *i*+*j*+*k*. egység teszteredményeire.

   4.3. Ha viszont nem talál hibátlan teszteredményt, az azt jelenti, hogy az elemzés során valamelyik hibátlannak feltételezett egység mind a *t* darab szomszédját hibásnak tesztelte. Ilyenkor a központi egység az adott sorban fennmaradó helyeket 0-val tölti ki, hiszen már „megtalálta” a lokális diagnosztikai képben a *t* korlát által „engedélyezett” összes hibás egységet.

   4.4. A folyamat addig tart, amíg vissza nem jutunk a kitöltés során a kiinduló *i*. egységnek megfelelő *i*. oszlophoz, vagyis amíg a teljes sor ki nincs töltve.

5. Amint a fenti séma szerint mind az *n* sor, azaz a teljes táblázat kitöltésre került, következik a processzorok hibaállapotának minősítése. Mit is értünk el eddig? Elkészült az *n* darab egységhez tartozó *n* darab lokális diagnosztikai kép (a táblázat sorai). Ezen lokális diagnosztikai képek között vannak jók (azok, amelyeknél a kiinduló *i*. egység valóban hibátlan volt), hiszen ezeket a megbízható teszteredmények alapján töltöttünk ki. Ennek megfelelően ezek a sorok megegyeznek. Ezen kívül vannak téves lokális diagnosztikai képek (azok, amelyeknél a kiinduló egység a valóságban hibás volt). Ezek a sorok többé-kevésbé véletlen értékeket tartalmaznak, és nem feltétlenül egyeznek meg egymással.

   Sajnos épp azt nem tudjuk, mely processzorok hibásak! Ezért a következőképpen járunk el: a B táblázatot függőlegesen, a B_k oszlopok szerint leolvasva megszámoljuk az 1-eseket (*i* szerint összegezzük a $b_{k,i}$ elemeket). Ha ez az összeg *t*-nél nagyobb, akkor az *i*. egység minősítése hibás lesz, ha az összeg legfeljebb *t*, akkor az *i*. egység hibátlan (a 3. ábra ezt az *n*-2. egységre mutatja).

(Ellenőrző kérdés: bizonyítsuk be, hogy a Meyer és Masson algoritmus a PMC tesztérvénytelenítésre vonatkozó egy lépében diagnosztizálhatóság feltételeinek teljesülése esetén helyesen működik!)

4 A mérés elvégzése

A mérés során egy központosított rendszerszintű diagnosztikai algoritmust kell elkészíteni egy C nyelvű program formájában, majd a kész programot futtatva megvizsgálni az algoritmus fontosabb jellemzőit. A feladatot megkönnyítendő rendelkezésre áll egy futtatókörnyezet, ami tartalmazza az olyan kiegészítő komponenseket, mint a *tesztelési gráf*, *hibainjektor* és *szindróma generálás* az adott tesztérvénytelenítési sémának megfelelően. Így a mérés során megírandó kódnak csak a diagnosztikai algoritmusra kell koncentrálnia és olyan formában készíthető el, mintha egy valós rendszer része lenne. A következő pont bemutatja a futtatókörnyezet tulajdonságait és útmutatóul szolgál a mérési program elkészítéséhez.

4.1 A diagnosztikai algoritmus megvalósítása

A mérésben elkészítendő a kiadott diagnosztikai algoritmus C nyelvű programkódja. Ez a kód a egy nagyobb program: a *futtatókörnyezet* részét képezi. Ebben a pontban ismertetjük a „programozói interfészt”, azaz azokat a változókat és függvényeket, amelyek kapcsolódásként szolgálnak a futtatókörnyezet többi moduljához. Ezeket kell felhasználnunk az algoritmus programozása során. A 4.3. fejezetben pedig megtalálható egy teljes (bár nagyon leegyszerűsített) diagnosztikai program kódja is. A javasolt módszer erre a példára, mint vázra felépítve elkészíteni az feladatként kapott algoritmus programját.

Az "`slidstat.h`" nevű header fájlban vannak a futtatókörnyezet többi modulja által közösen használt konstansok és típusdeklarációk. Ezért (az egyéb felhasznált standard header fájlok után) a diagnosztikai program elején *kötelező* beemelni az

```
#include "slidstat.h"
```

direktívával. A fájlban deklarált konstansok közül most csak a fontosabbakat ismertetjük, de a legalább egyszer érdemes végignézni az "`slidstat.h`" fájlt az összes lehetőség megismeréséhez. *A konstansok hibás alkalmazása vagy elhagyása a kész mérési programban nem várt problémákhoz vezethet*!

Az ismertetést a header fájlban való deklarálás sorrendjében kezdjük.

- A "`TRUE`" és "`FALSE`" konstansok logikai igaz és hamis értékként használhatók fel logikai kifejezésekben.
- Az "`OK`" és "`ERROR`" konstansok a végrehajtási státuszt visszaadó függvények (pl. a később ismertetett, általunk elkészítendő "`diag_init()`" és "`diagnosis()`" függvények) hibátlan és hibás végrehajtását jelölik.

- Igen fontos szerepet töltenek be a programban a "FAULT_FREE" és "FAULTY" konstansok. Ezek a hibátlan és hibás hibaállapotot jelölik mind a teszteredmények, mind az egységek minősítése esetén. Egyes diagnosztikai módszerek nem teljesek, azaz nem teszik lehetővé az összes egység hibaállapotának minősítését. Ilyen esetekben a "UNKNOWN" konstans jelölheti az ismeretlen hibaállapotot. A 4.3. fejezetben közölt program példát mutat arra, hogyan használhatóak fel ezek a konstansok a diagnosztikai algoritmusban.

A konstansok nagy része a futtató környezet paramétereihez kapcsolódik, azok felső határát illetve számát adja meg. Ezeket a paramétereket és konstansokat a 2. táblázat sorolja fel. (Az egyes paraméterek jelentésével részletesebben majd a felhasználói felület ismertetésekor foglalkozunk.) A táblázatban szerepel a konstans neve, értéke (azaz az adott paraméter maximális értéke) és a paraméter indítás utáni alapértelmezett értéke. Megtaláljuk itt a paraméter aktuális értékét tartalmazó változó nevét is.

2. táblázat: A paraméterekhez tartozó konstansok és változók

Név	Típus	Leírás	Konstans	Maximum	Kezdeti
nRounds	int	Futtatások száma	MAX_ROUND	65534	1024
nRepeats	int	Változatlan hibaminták száma	MAX_ROUND	65534	0
nProcs	int	Processzorok száma	MAX_PROC	32766	256
nNeighbors	char	Szomszédos processzorok száma	MAX_NEIGHBOR	8	4
fMax	int	Hibás processzorok száma	MAX_FAULT	32766	16
fgMax	int	Hibacsoportok száma	MAX_FGROUP	8094	0
iType	char	Tesztérvénytelenítés típusa	NUM_INVAL	9	0
nTopology	char	Tesztelési topológia típusa	NUM_TOPOLOGY	10	0
nInjector	char	Hibainjektálás típusa	NUM_INJECTOR	8	0

Ezeket a változókat szükség esetén felhasználhatjuk a diagnosztikai algoritmus kódjában is, bár a legtöbb esetben nincs rájuk szükség. Fontosak viszont az "nProcs" és "nNeighbors" változók. Az "nProcs" a rendszert alkotó egységek (processzorok) számát adja meg. Az egyes egységek azonosítói vagy indexei 0-tól indulnak és `nProcs – 1` értékig számozódnak. Az "nNeighbors" a tesztelési gráfban a szomszédos processzorok maximális számát definiálja. A szomszédok indexei szintén 0-tól indulnak és `nNeighbors – 1` értékig tartanak.

A fenti két változót az "extern" direktíva segítségével tehetjük elérhetővé az általunk írt kódban:

```
extern int nProcs;
extern char nNeighbors;
```

Az algoritmus megvalósításában központi szerepet játszik a szimulált rendszer adatait tartalmazó "psArray" nevű, "PROCSTAT" típusú, "nProcs" számú elemet tartalmazó tömb. Minden tömbelem egy feldolgozó egységet (processzort) jelképez. A szimulált rendszerben a processzorok azonosítása tehát a tömbben elfoglalt helyükkel, azaz a tömbindexükkel történik. Ezek az indexek 0-val indulnak és a legmagasabb index az `nProcs - 1`. A "psArray" tömb elemei struktúrák, amelyek a hozzájuk tartozó sorszámú processzor hibaállapotát, az általuk végzett tesztek eredményét, diagnosztikai minősítését és egyéb statisztikai adatait tárolják. A struktúra deklarációja a következő:

```
typedef struct procstat {
    char state;
    char inval;
    char tests[MAX_NEIGHBOR];
    char diag;
    int benign;
    int malign;
    int unknown;
} PROCSTAT;
```

Az alapvető konstansok, változók és struktúrák megismerése után lássuk hogyan használja fel ezeket és hogyan működik az elkészült, lefordított és teljes programmá összeszerkesztett futtatórendszer:

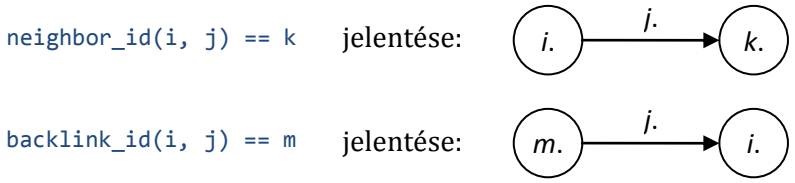
1. Az indulása után először az egységek hibaállapotát jelző "`psArray[i].state`" változót a hibátlan "`FAULT_FREE`" értékre állítja be, majd a beállított hibainjektálási módszer szerint hibás "`FAULTY`" egységeket generál.
2. A következő lépésben kitölti az egységek tesztérvénytelenítési modelljét tároló "`psArray[i].inval`" változót, majd a parancssori opciók között meghatározott tesztelési gráfnak megfelelően elvégzi a teszteket, és az eredményeket a "`psArray[i].tests[j]`" tömbelemekben (az *i*. tesztelő egység által a *j*. szomszédján elvégzett teszt) helyezi el.
3. Ezután futtatja a diagnosztikai rutint, amit a mérés során a mi készítettünk el. (Ez a lefordítás és a futtatókörnyezet kódjával való összeszerkesztés során vált a program részévé). *Fontos, hogy az általunk írt diagnosztikai rutin a visszatérése előtt helyezze el minden egység hibaállapotának minősítését a* "`psArray[i].diag`" *változóban*! A futtatókörnyezet összehasonlítja a diagnosztizált és a valós értéket, majd az eredményekből statisztikát készít.

Az általunk elkészítendő algoritmus tehát a "`psArray`" tömb segítségével, a "`psArray[i].tests[j]`" elemek olvasásával jut hozzá a teszteredményekhez, és ugyanebben a tömbben a "`psArray[i].diag`" változóban kell elhelyeznie a diagnosztika eredményét is. Mint említettük, a teszteredmények a tesztelési gráf szomszédossági viszonyainak megfelelően tárolódnak a tömbben. De honnan tudhatja a programunk, hogy mi az *i*. egység *j*. szomszédjának indexe a "`psArray`" tömbben? A „szomszédosság” fogalma ugyanis függ az éppen kiválasztott tesztelési elrendezéstől (topológiától). Erre a célra az alábbi két függvényt biztosítja a futtatókörnyezet (itt a programunkba beírandó deklarációkat adjuk meg):

```
extern int (*neighbor_id)(int iProc, char iNeighbor);
extern int (*backlink_id)(int iProc, char iNeighbor);
```

A "`neighbor_id(i,j)`" függvényhívás visszaadja az *i*. processzor *j*. szomszédjának azonosítóját. A rendszerre lefordítva ez annyit jelent, hogy az *i*. tesztelő egység teszteli a `neighbor_id(i,j)` indexű tesztelt egységet és a beállított tesztérvénytelenítésnek megfelelően „torzított” teszteredményt a futtatókörnyezet a "`psArray[i].tests[j]`" tömbelembe teszi.

A "`backlink_id(i,j)`" függvényhívás magyarázata egy kicsit bonyolultabb. Ez annak az egységnek az azonosítóját adja meg, akinek az *i*. processzor a *j*. szomszédja. A tesztelés fogalmaival leírva a `backlink_id(i,j)` indexű tesztelő egység teszteli az *i*. tesztelt egységet és az eredmény a "`psArray[backlink_id(i,j)].tests[j]`" tömbelemben található. A 4. ábra mindezt kissé szemléletesebben mutatja. Ezek után könnyű belátni, hogy `neighbor_id(backlink_id(i,j), j) == i`.



4. ábra: A "`neighbor_id(i,j)`" és "`backlink_id(i,j)`" függvények

A szomszédok számozása 0-val indul és a legmagasabb szomszéd index az `nNeighbors - 1`. A szomszédok sorrendje a tesztelési gráf topológiájának két- (esetleg három-) dimenziós képében vett irányokkal függ össze. Vegyük például az egyszerű (határolt) 3 × 3 processzorból álló kétdimenziós hálót, melynek képe egy négyzetrács. Itt a szomszédok száma `nNeighbors == 4` (kelet, észak, nyugat, dél). A 0. indexű egység a bal alsó sarokban helyezkedik el. Az irányokat az iránytű szerint jelölve a 0. processzor 0. szomszédja a tőle jobbra eső keleti 1-es, az 1. szomszéd északra (felfelé) a 3-as indexű processzor. Természetesen nem minden szomszéd létezik, példánkban a 0. egység 2. (nyugati, balra) és a 3. (déli, lefele) szomszédja *a háló határolt volta miatt* hiányzik. A hiányzó processzorok indexei helyett a függvények a "`NONEXIST`" értéket adják vissza. Az irányok számozását a 6. ábra adja meg.

A "neighbor_id(i,j)" és "backlink_id(i,j)" függvények felhasználását a 4.3. fejezetben közölt programból vett részlettel illusztráljuk. (A példában használunk egy "psClass" nevű hipotetikus struktúratömböt.) Azt mutatjuk be, hogyan lehet megszámolni egy processzorra a tesztelői által rajta elvégzett hibás lefutású tesztek számát. *Külön felhívjuk a figyelmet a teszteredmények és a hiányzó szomszédok kezelésére!*

```
for (i = 0; i < nProcs; i++) {
    for (j = fSum = 0; j < nNeighbors; j++) {
        if ((iTester = (*backlink_id)(i, j)) == NONEXIST)
            continue;
        result = (psArray[iTester]tests[j] == FAULTY) ? 1 : 0;
        psClass[i].fSum += result;
    }
}
```

Az általunk elkészítendő diagnosztikai algoritmus futtatása három fázisban történik: *előkészítő*, *diagnosztikai* és *lezáró* fázisban. A három fázisnak három C nyelven megírandó eljárás felel meg, melyek nevei a fázisokkal azonos sorrendben: "diag_init()", "diagnosis()" és "diag_close()".

Az előkészítő fázis a szimulált rendszer inicializálása során kapja meg a vezérlést, ekkor tehát az egyes processzorok adatai (hibaállapot, tesztérvénytelenítés, teszteredmények) még nincsenek beállítva. Az előkészítő fázis alatt ezekre nincs is szükség, ennek feladata ugyanis a diagnosztikai algoritmusban felhasznált változók, tömbök és struktúrák létrehozása, illetve kezdeti értékének beállítása.

Lássunk erre egy egyszerű példát! Nézzük meg, hogyan hozhatja létre és inicializálhatja az algoritmus az előző példában már használt "psClass" nevű dinamikus struktúratömböt. A struktúra többek között tartalmazza az előre és hátra irányú szomszédos processzorok indexeit tároló változók beállítását a már megismert függvényekkel:

```
int diag_init(peDiag)
char *peDiag;
{
    int i, j;
    if ((psClass = (PCLASS *)calloc(nProcs, PCSIZE)) == NULL) {
        fprintf(stderr, "diag_init(): allocation error!\n");
        exit(ERROR);
    }
    for (i = 0; i < nProcs; i++) {
        psClass[i].fSum = 0;
        for (j = 0; j < nNeighbors; j++) {
            psClass[i].fNB[j] = neighbor_id(i, j);
            psClass[i].bNB[j] = backlink_id(i, j);
        }
    }
    return (OK);
}
```

Az előkészítő fázis csak egyszer fut le a program indítása után. Ezzel szemben a diagnosztikai fázis annyiszor fut le, ahány szimulációs kísérletet, azaz ún. *diagnosztikai menetet* írtunk elő a program indításakor. A futtatókörnyezet a diagnosztikai meneteket egy ciklusban hajtja végre. Először hibátlanra állítja az összes egység hibaállapotát, majd hibákat injektál a szimulált egységekbe a kiválasztott hibainjektálási módszernek megfelelően. Ezután a processzorok tesztérvénytelenítésének megfelelően torzított teszteredményeket állít elő a tesztelési gráfnak megfelelően. A szindróma összegyűjtése (azaz a "psArray[i].tests[j]" tömbelemek feltöltése) után a felhasználó által elkészített diagnosztikai rutin kapja meg a vezérlést. (A diagnosztikai fázist megvalósító "diagnosis()" függvényre a 4.3. fejezetben találunk példát.) Miután a diagnosztikai program elvégezte a processzorok hibaállapotának minősítését (azaz a "psArray[i].diag" tömbelemek feltöltését), a futtató program összehasonlítja a diagnosztikai képet a valós hibaállapotokkal, és az eredményekből statisztikát készít.

A lezáró fázis a programból való kilépés előtt fut le egyszer, feladata a különböző dinamikus változók megszüntetése, esetleg saját statisztikák kiíratása, stb. Esetünkben a "diag_close()" eljárás egyetlen feladata a "psClass" dinamikus tömb törlése:

```
void diag_close()
{
    free(psClass);
    return;
}
```

A diagnosztikai modul létrehozása után le kell fordítanunk és össze kell szerkesztenünk a modult a futtatókörnyezettel. Erre GNU C fordító és "`bash`" shell használata esetén a "`make –f makefile.gnu`" parancsot használhatjuk. Így létrejön a "`diagnose`" program, aminek futtatásával ellenőrizhetjük a megvalósított diagnosztikai algoritmus működését, majd a paraméterek megfelelő változtatásával ki is értékelhetjük az algoritmusnak az adott jellemző(k)től függő viselkedését .

4.2 A futtatókörnyezet használata

Miután az előző pontban leírtak segítségével sikeresen megírtuk és lefordítottuk a mérési programot, következhet a megvalósított algoritmus kipróbálása és értékelése. Ehhez a fordítás végeredményeként kapott "`diagnose`" programot kell futtatnunk.

Amint az első futtatásnál látni fogjuk, a program csak, parancssoros felhasználói felülettel rendelkezik. A mérés során majd kiderül, hogy ez inkább előny, mint hátrány. Mivel a program elsőlegesen kötegelt (batch) használatra lett tervezve, indításakor a parancssorból olvassa be a paramétereit és a kimenetét a standard output-ra (és megadott fájlokba) írja. Meglehetősen sok paraméterrel rendelkezik (lásd 3. ábra), ezért arra is lehetőség van, hogy ezek egy részét, vagy akár mindet egy környezeti változó beállításával (neve: "`LID_OPT`") adjuk meg. Sőt, mindezen felül készíthető egy ún. specifikációs fájl is, melyben szöveges formában adhatjuk meg a paramétereket, így jól dokumentált és újrafelhasználható méréseket készíthetünk elő. A három paraméter megadási mód között a következő prioritás áll fenn:

környezeti paraméterek < parancssori paraméterek < specifikációs fájl

azaz a parancssori paraméterek felülbírálják a környezeti változóban megadott értékeket, azokat pedig felülírják a specifikációs fájlban szereplő paraméterek. A program induláskor kiírja az összes paraméter aktuális értékét, így ellenőrizhető, hogy az opciókat helyesen adtuk-e meg. Ugyanezek az értékek bekerülnek az ún. rendszerszintű és szelektív statisztikai fájlokba is (ha engedélyeztük létrehozásukat). Ezek segítségével később is visszakereshető, hogy milyen paraméterekkel végeztük az adott mérést.

3. táblázat: A futtató környezet parancssori opciói

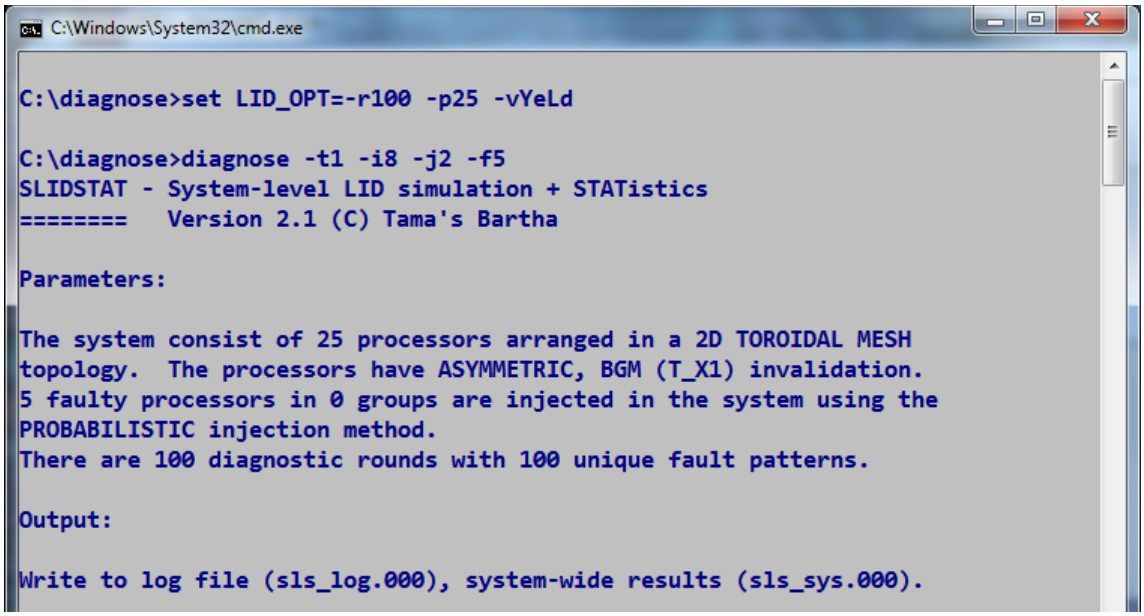
Opció	Paraméter	Leírás
`-r`	szám	A diagnosztikai algoritmus futtatásainak száma
`-q`	szám	Egymást követő változatlan hibaminták száma
`-p`	szám	Szimulált rendszert alkotó processzorok száma
`-f`	szám	Injektált hibás processzorok száma
`-g`	szám	Csoportos hibainjektálásnál hibacsoportok száma
`-i`	szám	Tesztérvénytelenítés típusa
`-t`	szám	Tesztelési gráf topológiájának típusa
`-j`	szám	Hibainjektálás típusa
`-xi`	szöveg	Extra paraméterek a tesztérvénytelenítéshez
`-xt`	szöveg	Extra paraméterek a tesztelési topológiához
`-xj`	szöveg	Extra paraméterek a hibainjektáláshoz
`-xd`	szöveg	Extra paraméterek a diagnosztikai algoritmusnak
`-ns`	szöveg	Specifikációs fájl neve
`-ny`	szöveg	Rendszerszintű statisztikai fájl neve
`-ne`	szöveg	Szelektív statisztikai fájl neve
`-nl`	szöveg	Részletes diagnosztikai kimenet fájl neve
`-v`	szöveg	Kimeneti fájlok megjelenítésének ki/be kapcsolói

Az egyes paraméterekhez tartozó parancssori opciókat a 3. táblázat ismerteti. Megadáskor az opciók sorrendje tetszőleges. A GNU programoknál megszokottal ellentétben viszont az opciót és a paraméter értékét egybe kell írni, pl. "`-p25 -f4`". A környezeti változóban megadott opciók formátuma azonos a parancssori opciókéval, így az előző példa Windows operációs rendszer alatt környezeti változóban a "`set LID_OPT=-p25 -f4`" paranccsal adható meg.

Hogy mindez érthetőbb legyen, lássunk egy részletesebb példát is a program indítására a következő paraméterekkel: a szimulált rendszer tartalmazzon 25 processzort 5 × 5 méretű kétdimenziós tórusz (nem határolt négyzetrács) topológiájú tesztelési gráfban (minden egység mind a négy szomszédját teszteli). A processzorok BGM tesztérvénytelenítéssel rendelkeznek, a rendszerbe $p = 1/5$ valószínűséggel injektálunk hibákat. A diagnosztikai algoritmust 100 alkalommal futtatjuk le egymás után. A kimeneti fájlok közül a napló és a rendszerszintű statisztikai fájl létrehozását engedélyezzük. A paraméterek egy részét a parancssorban, maradékát a környezeti változóban adjuk meg. (A példában UNIX operációs rendszert és "sh" shellt tételeztünk fel.)

```
# LID_OPT="-r100 -p25 -vYeLd"
# export LID_OPT
# diagnose -t1 -i8 -j2 -f5
```

A parancs hatására a mérési program elindul és az 5. ábra látható szöveget írja ki a képernyőre:



5. ábra: A futtatási paraméterek kiírása

A továbbiakban részletesen bemutatjuk a 3. táblázatban az egyes opciókhoz rendelhető paraméter értékeket. Az opciók első csoportja a szimulációhoz kapcsolódik. Az "`-r`" opció szolgál annak megadására, hányszor futtassuk le az algoritmust, azaz *hány diagnosztikai menet legyen*. A program ugyanis a diagnosztikai algoritmust nem csak egyszer futtatja le, hanem több alkalommal egymás után és az eredményekből statisztikát készít.

Alaphelyzetben minden menetben „tiszta lappal” indul a rendszer és új hibamintát generálunk, hiszen ez felel meg a való világnak. Statisztikailag azonban érdekes lehet vizsgálni, érzékeny-e az algoritmus a különböző szindrómákra azonos hibakészlet esetén. Ezért a "`-q`" opcióval megadhatjuk, *hány egymás után következő menetben ne változzon a hibaminta*. Ilyenkor csak a tesztérvénytelenítési sémában szereplő c = X vagy d = X értékek okozhatnak a szindrómában változatosságot.

A *szimulált rendszer méretét* a "`-p`" opcióval határozzuk meg. A program korrigálja a megadott értéket az adott topológiának megfelelően (pl. kétdimenziós háló topológia esetén, ha prímszámot adunk meg, akkor a program az ezt meg nem haladó legnagyobb négyzetszámot választja és egy négyzet alakú topológiát alakít ki). Az adott topológiához kapcsolódhatnak ún. extra paraméterek is, ezek szintén befolyásol(hat)ják az egységek számát. A már említett kétdimenziós háló topológiánál alaphelyzetben a háló négyzet alakú ("`-p25`" paraméter esetén 5 × 5 méretű), de ún. *extra paraméterekkel* (lásd később) megadhatjuk a hosszúságot és szélességet is, és így téglalap alakú területet is létrehozhatunk. Pl. ha egy

27 processzorból álló téglalap alakú 3 × 9-es hálót akarunk létrehozni, akkor az `x` és `y` extra paraméterek segítségével adhatjuk meg a háló szélességét és hosszúságát: "`-p27 –xtx3:y9`".

Az "`-f`" opció a rendszerbe injektált hibás egységek számát határozza meg. Ez csak egy várható érték, függ a kiválasztott hibainjektálási módszertől. A hibák véletlenszerűen generálódnak, így a megadott hibaszámtól lefelé és felfelé is eltérhet a hibás egységek tényleges száma. Kivétel ez alól a *deterministikus hibainjektálás* (lásd később), amikor mi határozzuk meg, hogy melyik processzor hibás.

A valóságot a teljesen egyenletes eloszlásban véletlenszerűen elterített hibák nem írják le pontosan. A hibás processzor ugyanis a környezetét is „megfertőzheti”. Ez a *hibaterjedés* jelensége. Ennek a jelenségnek a szimulálásához lehetőség van ún. *hibacsomók* létrehozására a "`-g`" opcióval. Például a "`-f10 -g2`" paraméterek jelentése: a hibák két csoportban tömörülnek, csoportonként átlagosan 5-5 hibás processzorral. A hibacsomók számát csak akkor van értelme megadni, ha egy csoportos hibákat generáló hibainjektálás típust választunk, egyébként a program a paramétert figyelmen kívül hagyja.

Itt hívjuk fel a figyelmet arra, hogy a program *nincs felkészítve az egymásnak ellentmondó paraméterek figyelésére* (pl. ha a specifikált rendszerméret és a topológia extra paraméterből következő processzorszám nem egyezik, vagy determinisztikus hibainjektálásnál a megadott hibaszám és a specifikációs fájlban szereplő hibák száma különböző), *ezért erre különösen figyelni kell*!

Az eddig látott, első csoportba tartozó paraméterek megadási módja egyértelmű, a további paraméterek azonban speciális, táblázatban megadott értékeket használnak. Ezek közül elsőként a "-i" opcióval megadható *tesztérvénytelenítési modelleket* vesszük sorra, amelyeket a 4. táblázat ír le.

Az értékek 0-8 között az általános tesztleképzésbe tartozó kilenc alapvető modellt jelentik. Ezeket megadva a rendszer homogén lesz, minden egység az adott tesztérvénytelenítési modellt követi. Speciális algoritmusoknál érdekes lehet megvizsgálni a működést heterogén környezetben is, erre szolgál a 9-es és 10-es érték. Az 9-es értékkel a program minden processzorra egy adott halmazból véletlen módon választ tesztérvénytelenítést. A 10-es érték hatása átmenet a homogén és heterogén eset között: a program létrehoz néhány homogén tartományt, és minden ilyen homogén tartományban az összes egységhez azonos, de véletlenszerűen választott modellt rendel.

4. táblázat: Tesztérvénytelenítés típusok

Típus	Tesztérvénytelenítési modell	Jelölés	Konstans
0	szimmetrikus érvénytelenítés, PMC	TI_{XX}	`TI_XX`
1	0-fail-safe tesztelő	TI_{00}	`TI_00`
2	„lovag”, perfekt tesztelő	TI_{01}	`TI_01`
3	szekvenciális tesztelő	TI_{0X}	`TI_0X`
4	„lókötő”, konzekvens hazudozó	TI_{10}	`TI_10`
5	irreflekszív érvénytelenítés, HK2	TI_{11}	`TI_11`
6	reflekszív érvénytelenítés, HK1	TI_{1X}	`TI_1X`
7	következetlen hazudozó	TI_{X0}	`TI_X0`
8	aszimmetrikus érvénytelenítés, BGM	TI_{X1}	`TI_X1`
9	egyenletes eloszlású véletlen érvénytelenítés		
10	homogén blokkokban véletlen érvénytelenítés		

A tesztérvénytelenítésnél lehetőség van ún. *extra paraméterek* megadására is. A már látott extra paraméterek olyan opciók, amik a standard paraméterek „finomhangolását” teszik lehetővé. Megadási módjuk speciális: egy közös kapcsolóval (pl. a tesztérvénytelenítés extra paraméterei esetén a "`-xi`" kapcsolóval) több paraméter is megadható egyetlen folyamatos szövegként, az egyes paramétereket egymástól valamilyen (az adott extra paramétertől függő) határoló karakterrel elválasztva.

A tesztérvénytelenítés extra paramétereinek csak a 9-es és 10-es heterogén modellek esetén van jelentése. Ezen esetekben arra adnak lehetőséget, hogy definiáljuk azt a modell halmazt, amelyből a program véletlenszerűen választani fog. Ehhez a "`-xi`" kapcsoló után fel kell sorolnunk a kiválasztott tesztérvénytelenítési modelleket (két karakteres rövidítésükkel jelölve) egymástól pontokkal

elválasztva. Pl. a "`-i9 -xiXX.X1.11`" opció eredményeként egy olyan heterogén rendszer jön létre, amiben az egységek tesztérvénytelenítése a PMC, BGM és a HK2 modellek közül kerül ki.

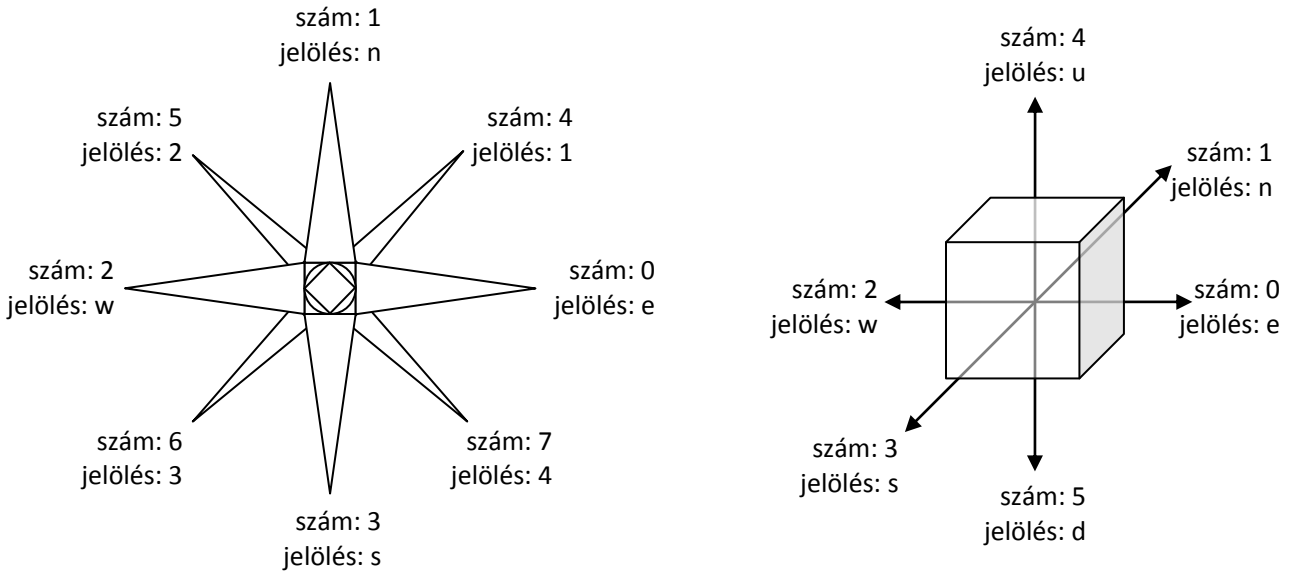
A tesztek elvégzésének menetét, a tesztelő → tesztelt egység kapcsolatokat a tesztelési gráf határozza meg. Ennek az irányított gráfnak a *topológiáját* adhatjuk meg a "`-t`" opcióval. A jelenleg rendelkezésre álló topológia típusokat az 5. táblázat sorolja fel. Ezek a szakirodalomból jól ismert és az elosztott rendszerekben gyakran használt összeköttetési hálózatokon, ezért nem térünk ki a bemutatásukra.

A táblázat megadja az adott tesztelési gráf típusban értelmezett irányok jelölését is. Mint a programozásról szóló részben már említettük, az irányokat a topológia két- vagy háromdimenziós képén vett térképészeti irányok angol megfelelőinek (East, North, West, South) kezdőbetűivel jelöljük. Háromdimenziós hálók esetén ehhez jön hozzá a függőleges tengelyen mért két irány (Up, Down).

5. táblázat: Tesztelési gráf topológia típusok

Típus	Tesztelési gráf topológia	Irányok	Extra paraméter
0	Kétdimenziós határolt háló	`e,n,w,s`	`x,y,d`
1	Kétdimenziós tórusz háló	`e,n,w,s`	`x,y,d`
2	Kétdimenziós áthajtott tórusz háló	`e,n,w,s`	`x,y,d`
3	Háromdimenziós határolt háló	`e,n,u,w,s,d`	`x,y,z,d`
4	Háromdimenziós tórusz háló	`e,n,u,w,s,d`	`x,y,z,d`
5	*k*-ad fokú bináris hiperkocka		`k`
6	bináris fa		
7	$D_{\sigma,A}$ gyűrű		`s,A`
8	3-ad fokú szisztolikus tömb	`e,n,w,s`	`x,y,d`
9	6-od fokú szisztolikus tömb	`e,n,w,s,1,2,3,4`	`x,y,d`

A 6-od fokú szisztolikus tömb ugyan kétdimenziós struktúra, de a négy alapirányhoz négy „ferde” irány jön hozzá, melyeket az egyszerűség kedvéért csak számokkal (és nem kétbetűs rövidítéssel, pl. `nw` a NorthWest megfelelőjeként) jelölünk. A többi topológiában ilyen geometriai megfeleltetés nem lehetséges, azokban a szakirodalomban szokásos irányítást valósítottuk meg. Az irányokat és jelölésüket szemléletesen a 6. ábra mutatja be.



6. ábra: Topológiai irányok száma (a "`neighbor_id(i,j)`" és "`backlink_id(i,j)`" függvényekhez) és jelölése

A tesztelési topológiához is léteznek extra paraméterek (ezekre már utaltunk a "`-p`" opció leírásakor). A topológia extra paramétereit a "`-xt`" kapcsoló jelöli, az egyes paramétereket egymástól kettőspont választja el. Az egyes topológiákra értelmezett extra paramétereket szintén az 5. táblázatban találjuk.

Lássunk erre rögtön egy példát! Ha a programot a "`-p35 -t1`" paraméterekkel indítjuk, akkor a szimulált rendszer automatikusan 5 × 5 egységet tartalmazó, kétdimenziós tórusz háló topológia lesz, mert alaphelyzetben a program négyzet alakúra választja a háló formáját. Ha tehát egy tényleg 35 processzorból álló, téglalap alakú 7 × 5-ös hálót akarunk szimulálni, akkor a következőképpen adhatjuk meg a háló szélességét és hosszúságát: "`-p35 -t1 -xtx7:y5`".

Extra paraméterekkel arra is lehetőségünk van, hogy csak bizonyos kiválasztott irányokat engedélyezzünk. Folytatva előző példánkat: alapértelmezés szerint mind a négy irány be van kapcsolva. Tehát a szomszédos egységek kölcsönösen tesztelik egymást, hiszen pl. egy adott egység keleti szomszédjának ő a nyugati szomszédja. Szüntessük meg a kölcsönös teszteket úgy, hogy a processzorok csak a keleti és északi szomszédjukat teszteljék! Erre a `d` extra paraméter szolgál, így a példa teljes paraméterezése: "`-p35 -t1 -xtx7:y5:den`".

A következő csoportot a hibainjektáláshoz kapcsolódó paraméterek alkotják. A *hibainjektálási módszert* a "`-j`" opcióval határozhatjuk meg. Az elérhető típusokat a 6. táblázat ismerteti. A módszerek két csoportba oszthatók: *véletlen* és *determinisztikus* (a felhasználó által megadott) hibainjektálásra (ez utóbbival részletesen foglalkozunk később a specifikációs fájl kapcsán). A véletlen hibainjektálás szintén kettéválasztható egyenletesen elterített és csoportos hibák esetére.

6. táblázat: Hibainjektálás típusok

Típus	Hibainjektálás módja
0	adott számú, egyenletes eloszlású hiba
1	determinisztikus (specfikáció fájlban megadott) eloszlás
2	adott valószínűséggel meghibásodó egységek
3	hiba környékén nagyobb valószínűséggel meghibásodó egységek
4	lineáris eloszlású csoportos hiba
5	kétdimenziós eloszlású csoportos hiba
6	lineáris eloszlású hiba több csoportban
7	kétdimenziós eloszlású hiba több csoportban

Az *egyenletes hibainjektálás* a külső környezet vagy az öregedés hatásait hivatott modellezni. Kétféle egyenletes hibageneráló módszert tartalmaz a futtató környezet:

1. Az egyik eset közelebb áll a fizikai modellhez: minden egység adott (a specifikált hibaszámból kiszámított) valószínűséggel lesz hibás, egyébként hibátlan marad.
2. A másik esetben adott számú, egyenletes eloszlású véletlen indexet generál a program: ezek lesznek a hibás processzorok.

Mindkét esetben eltérhet a valós hibaszám a parancssorban megadottól. Az első módszernél ez magától értetődik, hiszen ha minden egység $1/p$ valószínűséggel hibásodik meg, akkor a specifikált hibaszám a valós hibaszám várható értéke lesz. A második esetben az eltérés oka más. Ha a hibák száma a rendszer méretéhez képest relatíve nagy, a második módszer használatakor megnő annak valószínűsége, hogy egy már korábban hibássá tett egység azonosítóját sorsoljuk ki újra. Ha ennél a módszernél addig próbálkoznánk, amíg végre elérjük a kívánt hibaszámot, nagyon megnőhetne a hibageneráláshoz fordított idő és lassú lenne a program futása. Ezen több módon is segíthetünk. A futtatókörnyezet egy egyszerű megoldást használ: ha a hiábavaló próbálkozások száma meghalad egy adott korlátot (túl sokszor jön ki a már korábban hibássá tett egység azonosítója), akkor egyszerűen „feladja”, és eggyel csökkenti az előállítandó hibák számát. Így a második hibainjektáló módszernél a generált hibák száma a specifikálttól csak lefele térhet el.

A *csoportos hibák* a hibaterjedés hatását modellezik. Ilyenkor a hibás egység környezetében nagyobb valószínűséggel hibásodnak meg további egységek. Hasonlóan az egyenletes eloszlású hibageneráláshoz itt is kétféle módszer áll rendelkezésünkre:

1. Az elsőnél véletlenszerűen generálunk egy hibás processzort, majd az indexekre egyenletes eloszlás torzításával a hiba környezetében nagyobb valószínűséggel választunk újabb hibás egységeket. Ezt végezhetjük a topológiától függetlenül, az indexekre nézve „lineárisan”, amikor az *i*. egység közvetlen környezete az *i*+1. és *i*-1. egységet jelenti. Sajnos az indexek szomszédossága általában nem jelent topológiai szomszédosságot, így a lineáris eloszlású csoportos hiba össze-vissza szétszórva helyezkedhet el a szimulált rendszerben. Ezért külön a kétdimenziós hálókra létezik topológiai szomszédosságot használó hibainjektáló módszer is.
2. A csoportos hibainjektálás másik, fizikai modelljénél nem azonosítókkal dolgozunk, hanem egyszerűen megnöveljük az egységek meghibásodási valószínűségét a hibák szomszédságában. Itt a szomszédosságot topológiai értelemben használjuk, így az előbbi probléma nem lép fel.

Mint említettük, nem csak a parancssor szolgálhat a futtató környezet paramétereinek beállítására. A másik lehetőséget az ún. *specifikációs fájl* szolgáltatja. Ez egy szöveges formátumú fájl, aminek szintaktikája a Windows rendszereken használt `INI` konfigurációs fájlokéhoz hasonlít. Két fő eleme van: a szögletes zárójelek között szereplő *szekciónevek*, amelyek a logikailag összetartozó paramétereket csoportosító szekciókat jelölik, valamint a *paraméter értékadások*. A futtató környezet által elfogadott szekció- és paraméterneveket és azok jelentését a 7. táblázat sorolja fel.

7. táblázat: A specifikációs fájl parancsai

Név	Paraméter	Leírás
[system] szekció (rendszerparaméterek)		
`processors`	szám	Szimulált rendszert alkotó processzorok száma
`faults`	szám	Injektált hibás processzorok száma
`fault groups`	szám	Csoportos hibainjektálásnál a hibacsoportok száma
`invalidation`	szám	Tesztérvénytelenítés típusa
`test rate FG`	szám	HIBÁS JÓ tesztek hibás lefutásának aránya
`test rate FF`	szám	HIBÁS → HIBÁS tesztek hibás lefutásának aránya
`topology`	szám	Tesztelési gráf topológiájának típusa
`injector`	szám	Hibainjektálás típusa
[faults] szekció (determinisztikus hibainjektálás leírása)		
`faulty`	lista	Hibás processzorok azonosítóinak listája
[selective] szekció (szelektív statisztika megadása)		
`processors`	szám	Szelektíven figyelt processzorok száma
`watch`	lista	Szelektíven figyelt processzorok azonosítóinak listája

A paraméter értékadások kétféle megadási módja (attól függően, hogy *egyszerű* vagy *lista* paraméterről van szó):

```
paraméternév = érték
paraméternév = érték1, érték2, …
```

Mielőtt elmélyednénk ennek a táblázatnak a tanulmányozásában, lássunk egy konkrét példát specifikációs fájlra (a paraméter értékek megadhatók decimális és hexadecimális alakban is):

```
[system]
    processors = 16
    faults = 4
    fault groups = 1
    topology = 1
    injector = 1
[faults]
    faulty = 0x0A, 0x0B
    faulty = 0x0E, 0x0F
[selective]
    processors = 16
    watch = 0x09, 0x0A
    watch = 0x0D, 0x0E
```

Tegyük fel, hogy a fenti példa specifikációs fájlt "`example.psf`" néven mentettük el! Ha a programunkat a "`-r100 -nsexample.psf`" opciókkal indítjuk, akkor 100 diagnosztikai menetben futtatjuk az általunk megvalósított diagnosztikai algoritmust, a szimulált rendszer egy 16 processzorból álló 4 × 4-es kétdimenziós tórusz háló topológiájú rendszer lesz, amelyben determinisztikusan (a hibás egységek azonosítóit egy listában megadva) 4 egységet rontunk el. Definiálunk ún. *szelektíven figyelt* processzorokat is, ezekre az egységekre vonatkozóan a *szelektív statisztikai fájlba* külön kigyűjtés készül.

A program kétféle kimenetet készít. Egyrészt a képernyőn követhetjük a program futását, másrészt kimeneti fájlok készülnek. Az 5. ábra már mutatott erre egy példát: a képernyőkimenetnek azt a részét, melyet az indítás után ír ki a program. A 7. ábra azt a szöveget mutatja, amit a program a diagnosztikai menetek során az általunk megírt diagnosztikai modul végrehajtása után ír ki. (Az összes példa képernyőkép ugyanazon futtatás során kiírt kimenet különböző részeit mutatja.)

```
Output:

Write to log file (sls_log.000), system-wide results (sls_sys.000).

DIAGNOSE - SLIDSTAT diagnostic module
--------   Version 1.0

Starting diagnostic simulation:

(00%) .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0
(24%) .0 .+ .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0
(48%) .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0
(72%) .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .0 .+ .0 .+ .0 .0 .0 .0
(96%) .0 .0 .0 .0

Diagnostic simulation complete.
```

7. ábra: A program futása közben kiírt információk

Emlékezzünk vissza, hogy az akkori példánkban 100 diagnosztikai menetet írtunk elő. A képernyőképen egy rövid szöveget látunk, amelyet az általunk megírt diagnosztikai program (lásd 4.3. fejezet) ír ki, majd a futtató környezet közli velünk, hogy elkezdi futtatni a diagnosztikai algoritmust. A futás során zárójelek között az addig elkészült munka százalékos állásáról kapunk információt. Ezután egy pontból és egy karakterből álló sorozatok jönnek. A pont a diagnosztikai rutin elindítását jelenti, az ezt követő karakter az algoritmus futási eredményeiről tájékoztat:

- "**0**" esetén a diagnosztizált és a valós hibaállapot megegyezik,
- "**+**" esetén volt olyan hibátlan egység, akik tévedésből hibásnak minősítettünk,
- "**-**" esetén hibásat minősítettünk hibátlannak, és
- "**#**" esetén mindkét típusú tévedés előfordult.

A parancssorban megadott számú diagnosztikai menet után a program statisztikát készít a futtatások eredményeiből, majd kilép. A statisztikai kimenet a 8. ábra látható.

A képernyőn kívül fájlokba is kiírja a fontosabb adatokat a futtató környezet. A fájlok készítését a "`-v`" opció szabályozza. Az opció után a kimeneti fájlokra vonatkozó kapcsoló karaktereket írhatunk, amiket a 8. táblázatban találunk meg. A karakter kisbetűs változata kikapcsolja, nagybetűs változata bekapcsolja az adott kimeneti fájl készítését. A karakterek sorrendje tetszőleges. Tehát pl. a "`-vLYed`" opció jelentése: készül napló és rendszerszintű statisztikai fájl, viszont nem készül szelektív statisztikai és diagnosztikai kimenet fájl. A fájlok nevét mi is megadhatjuk (a 8. táblázatban felsorolt paraméterek segítségével), ha viszont ezt nem tesszük, akkor a futtató környezet maga fabrikál nekik nevet az alapértelmezett névből és egy számlálóként funkcionáló kiterjesztésből.

```
Diagnostic simulation complete.

Fault injection
statistics:     +---------+----------------+---------+-----------+---------+
                | Minimum |  Total number  | Average | Deviation | Maximum |
Fault-free      +=========+================+=========+===========+=========+
processors      |    16   |   2023 (81%)   |   20.2  |     1.8   |    24   |
Faulty          +---------+----------------+---------+-----------+---------+
processors      |     1   |    477 (20%)   |    4.8  |     1.8   |     9   |
                +---------+----------------+---------+-----------+---------+

System-wide
results:        +---------------+---------+---------+-----------+----------+
                |  Total number | Maximum | Average | Deviation | Interval |
Misdiagnosed    +===============+=========+=========+===========+==========+
  fault-free    |     4 ( 1%)   |     2   |   0.0   |     0.2   |    0.0   |
Misdiagnosed    +---------------+---------+---------+-----------+----------+
  faulty        |     0 ( 0%)   |     0   |   0.0   |     0.0   |    0.0   |
                +---------------+---------+---------+-----------+----------+
Unidentified    |     0 ( 0%)   |     0   |   0.0   |     0.0   |    0.0   |
                +---------------+---------+---------+-----------+----------+

There were 3 ( 3%) incorrect and 0 ( 0%) incomplete diagnostic rounds.
Misdiagnosed fault-free processors in 3 rounds.

The statistical parameters were calculated using a sample of 100 results.
The confidence interval of the parameters has a 81% coefficient.
```

8. ábra: A diagnosztikai algoritmus eredményeiből képzett statisztika

8. táblázat: Kimeneti fájlok

Leírás	Kapcsoló	Alapértelmezett név	Konstans	Változó
Napló fájl	l/L	`sls_log`	`SLS_LOG`	`pLN`
Rendszerszintű statisztikai fájl	y/Y	`sls_sys`	`SLS_SYS`	`pYN`
Szelektív statisztikai fájl	e/E	`sls_sel`	`SLS_SEL`	`pSN`
Diagnosztikai kimenet fájl	d/D	`sls_diag`	`SLS_DIAG`	`pDN`

A kimeneti fájlok közül számunkra a diagnosztikai modul programozása során a *napló fájl* a fontosabb. Ebben az algoritmus futtatásával kapcsolatos, másutt meg nem jelenő fontos információk kaptak helyet. A fájl formátuma szöveges, de eléggé egyedi, megalkotásakor ugyanis a gépi feldolgozás egyszerűsége volt az elsődleges szempont. Szemléltetésként álljon itt rövid részlet egy napló fájlból:

```
; pattern 2: 5 fault(s) in 0 group(s)
[12] { 2 5 0
    0009 000C 000E 0011 0012
}
; maximal benign misses (system-level): 1 (0)
; maximal malign misses (system-level): 1 (0)
; misdiagnosed processors! benign: 1, malign: 1, unknown: 0
[13] { 1 1 0
    000D+ 0011-
}
```

A "pattern" kezdetű szekcióban a szögletes zárójelek között az adott diagnosztikai menet sorszámát, a kapcsos zárójelek között pedig a hibás egységek azonosítóinak hexadecimális értékét találjuk. A "misdiagnosed processors" kezdetű szekcióban a kapcsos zárójelek között a hibásan minősített egységek azonosítóinak hexadecimális értéke található. A kódot követő karakter a tévedés fajtájára utal.

4.3 Programozási példa

Ebben a részben bemutatunk egy egyszerű „rendszerszintű központosított valószínűségi diagnosztikai" programot. A példa célja segítséget adni a feladat megoldásához a futtató rendszerhez való kapcsolódás bemutatásával, és egy olyan „vázat" adni, amelyre „felhúzhatjuk" az elkészítendő mérési feladatot.

A példaprogram által megoldott diagnosztikai feladat leírása vázlatosan a következő: a kétdimenziós tórusz topológiájú hálózatban minden feldolgozó egység teszteli mind a négy szomszédját. Ezek után minden egységnél összesítjük azokat a *hibás teszteket*, amelyeket az *tesztelői az adott egységről* hoztak létre. Ez az összeg ad egy közelítést arra nézve, hogy mennyire „gyanús" az adott egység, mekkora valószínűséggel hibás. Amennyiben ez a szám meghalad egy előre meghatározott limitet, akkor az adott egységet rossznak minősítjük, ha nem, akkor jónak. A limitet egyszerűen a szomszédok számának felére választjuk, azaz ha a tesztelők többsége rossznak látta az adott processzort, akkor mi is annak minősítjük. Természetesen ez a módszer elég durva becslést tud csak adni az egységek valós hibaállapotára, hiszen a tesztelők maguk is hibásak lehetnek, és ha a hibás tesztelők az adott egység környezetében többségben vannak, akkor a jó egységet is tévesen rossznak minősíthetjük. Ezért érdekes lesz majd a futtatásokból nyert statisztikai adatok vizsgálata.

A diagnosztikai modult logikailag összetartozó részekre bontva ismertetjük. A teljes kódot megadjuk, tehát a darabokat összeillesztve az egész modult megkapjuk. Mivel a példa modul forráskódja szövegfájlként is hozzáférhető, annak tanulmányozása is segítheti az áttekintést.

A program a C nyelv szokása szerint a felhasznált függvények prototípusait definiáló és konstansokat deklaráló header fájlok beemelésével indul.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

#include "slidstat.h"
```

A header fájlok feldolgozása után deklarálhatjuk a lokálisan használt konstansokat. Esetünkben egyetlen lokális konstansra van szükség, ami a az adott egységre vonatkozó hibás tesztek maximális értékét határozza meg (ezen összeg felett az egységet hibásnak minősítjük):

```
#define LIMIT (nNeighbors/2)
```

A szimbólumok deklarálása végeztével a külső (más modulokban, azaz esetünkben a futtató-környezetben deklarált) változók és függvények felsorolása következik. Az "nProcs" és "nNeighbors" változók szerepéről már volt szó a programozói felület ismertetésekor. A két új külső változó: "verbose" és "pYN" azért nem került ott megemlítésre, mert szerepük nem lényeges a mérési feladat szempontjából. A "verbose" bináris flageket tartalmaz, melyek megmutatják, mely kimeneti fájlok készítését engedélyeztük a parancssorban. A "pYN" változó pedig a rendszerszintű statisztikai fájl nevét

tartalmazza. (A flagek jelölése és a különböző kimeneti fájlok nevei szerepelnek a felhasználói felületet ismertető pontban a 8. táblázatban.)

```
extern char verbose;
extern int nProcs;
extern char nNeighbors;
extern char pYN[];
```

A futtatókörnyezet által rendelkezésre bocsátott topológiát kezelő függvények szintén bemutatásra kerültek már. Mivel ezek külső függvények, ezért ezeket következő módon, az "`extern`" direktívával emelhetjük be a "`diagnose.c`" programba:

```
extern int (*neighbor_id)(int iProc, char iNeighbor);
extern int (*backlink_id)(int iProc, char iNeighbor);
```

Most deklaráljuk a program egyetlen globális változóját: a hibahatár értékét tároló "`limit`" változót. Azért használunk erre a feladatra egy változót is (a "`LIMIT`" konstans mellett), hogy a hibahatárt a parancssorban egy extra paraméter segítségével akár át is állíthassuk.

```
static int limit;
```

A konstansok és változók deklarálása után következhet az előkészítő fázis "`diag_init()`" eljárásának kidolgozása. Ebben az egyszerű példában nem használunk dinamikus változókat és inicializálnunk sem kell semmit. Viszont példát mutatunk a futtató környezet egy eddig nem ismertetett lehetőségére: a diagnosztikai extra paraméterek használatára. Ezeket a parancssorban a "`-xd`" opcióval adhatjuk meg. A futtató környezet a diagnosztikai paramétereket nem dolgozza fel, egyszerűen szövegként bemásolja a kapcsoló mögötti részt a "`peDiag`" nevű változóba (ez változó a "`diag_init()`" eljárás egyetlen átadott paramétere). A "`peDiag`" változó tartalmát nekünk kell feldolgoznunk, ez leshető el az alábbi részletből. Az extra paraméter feldolgozásán túl néhány üzenetet írunk ki a képernyőre és kiírjuk a diagnosztikai extra paramétereket a rendszerszintű statisztikai fájlba (mindkét tevékenység opcionális):

```
int diag_init(peDiag)
char *peDiag;
{
    FILE *pYF;
    if (*peDiag == '\0')
        limit = (int)LIMIT;
    else if (sscanf(peDiag, "l%d", &limit) < 1 || limit > nNeighbors) {
        fprintf(stderr, "diag_init(): invalid parameter format!\n");
        exit(ERROR);
    }
    printf("DIAGNOSE - SLIDSTAT diagnostic module\n");
    printf("--------   Version 1.0\n\n");
    if (strlen(peDiag) > 0)
        printf("Extra parameters for diagnostics '%s'.\n\n", peDiag);
    if ((verbose & SLS_SYS) != 0) {
        if ((pYF = fopen(pYN, "a")) == NULL) {
            fprintf(stderr, "diag_init(): file open error [%s]!\n", pYN);
            exit(ERROR);
        }
        fprintf(pYF, "";dextra\n"");
        fprintf(pYF, ""%14s\n\n"", (strlen(peDiag) > 0) ? peDiag : "--");
        fclose (pYF);
    }
    return (OK);
}
```

Példánkban a hibalimit értékét adhatjuk meg extra paraméterként. Ha a limitet 3-ra akarjuk állítani, akkor a parancssorba az "`-xdl3`" opciót kell írnunk (ekkor a "`peDiag`" szövegváltozó értéke "`l3`" lesz).

Az előkészítő rész után következhet maga a diagnosztikai rutin. Ennek két paramétere van: a szimulált rendszer adatait tartalmazó "`psArray`" tömb, valamint az aktuálisan végrehajtott diagnosztikai menet sorszámát tartalmazó "`iRound`" nevű változó. Az utóbbi adatra pl. saját statisztikák készítésénél vagy hibaüzenetek kiírásakor lehet szükség, a mostani példában nem használjuk fel.

A diagnosztikai rutin működése röviden a következő:

- minden *i*. processzor esetében megvizsgáljuk az őt tesztelő összes *j*. szomszédját (az "`iTester`" változó tartalmazza a tesztelő processzor azonosítóját),
- megszámoljuk, hogy a tesztelők közül hányan találták az *i*. processzort hibásnak, és
- ha ennek az "`fSum`" változóban gyűjtött értéke meghaladja a "`limit`" értékét, akkor a processzort hibásnak minősítjük, egyébként hibátlannak,
- a minősítést, azaz a diagnosztizált hibaállapotot beállítjuk a "`psArray[i].diag`" változóban.

A diagnosztikai rutin lényege tehát minden egyes egységre a "`psArray[i].diag`" változó beállítása az algoritmus által kiszámított minősítésre. Ismét felhívjuk a figyelmet arra, hogy ezt a változót a futtatókörnyezet *nem inicializálja az egyes diagnosztikai menetek elején, ezt nekünk kell megtennünk*! (A példában nincs szükség a "`psArray[i].diag`" változó inicializálására, mert biztosított, hogy minden egységre beállítjuk az értékét.) Az egyik leggyakoribb elkövetett hiba, hogy egyes egységek esetén a "`psArray[i].diag`" változó kitöltetlen marad, emiatt ott egy korábbi téves érték marad benn és ez az algoritmus hibás működéséhez vezet.

```
int diagnosis(psArray, iRound)
PROCSTAT psArray[];
int iRound;
{
    int i, j;
    int iTester;
    int fSum;
    for (i = 0; i < nProcs; i++) {
        for (j = fSum = 0; j < nNeighbors; j++) {
            if ((iTester = (*backlink_id)(i, j)) == NONEXIST)
                continue;
            fSum += (psArray[iTester].tests[j] == FAULTY) ? 1 : 0;
        }
        psArray[i].diag = (fSum > limit) ? FAULTY : FAULT_FREE;
    }
    return (OK);
}
```

Az utolsó megírandó kódrészlet a lezáró fázist megvalósító "`diag_close()`" függvény. Mivel dinamikus memóriát nem allokáltunk és saját statisztikát sem készítettünk, melynek eredményét itt írathatnánk ki, a "`diag_close()`" törzse üres (de el nem hagyható, hiszen akkor nem tudnánk összeszerkeszteni az általunk írt programot a futtatókörnyezet többi moduljával).

```
void diag_close()
{
}
```

4.4 A mérés kiértékelése

A mérés kidolgozása során tehát először is meg kell értenünk a kiadott diagnosztikai algoritmus működését, majd meg kell valósítanunk az algoritmust C nyelven. Ezután lefordítjuk a diagnosztikai modult és összeszerkesztjük a futtatókörnyezet kódjával, így egy futtatható programot kapunk.

Amikor az algoritmus működése számunkra helyesnek tűnik és különböző méretű, topológiájú, hibaszámú és tesztérvénytelenítésű rendszerekben is stabilan működik, mutassuk be a programot a mérésvezetőnek! Ha a mérésvezető is megfelelőnek ítéli a programot, akkor elkezdhetjük a kész diagnosztikai algoritmus vizsgálatát. Elsőként mindenképp érdemes megfigyelni a diagnosztikai pontosság alakulását adott méretű rendszerben növekvő hibaszámnál. Egy ilyen vizsgálat eredményét mutatja a 4.3. fejezetben közölt példa algoritmus futtatásakor 10 × 10-es tórusz topológiájú rendszerben PMC tesztérvénytelenítés mellett az első oszlopban látható növekvő hibaszámra a 9. táblázat.

9. táblázat: A 4.3. fejezetben bemutatott példa algoritmus viselkedése a hibás egységek számának függvényében

hibaszám	incorr	incomp	benign	malign	mgtot	mgmax	mgavg	mftot	mfmax	mfavg
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
4	14	0	2	12	2	1	0,02	12	1	0,03
10	154	0	30	132	32	2	0,03	155	4	0,15
25	913	0	416	833	572	4	0,56	2116	10	2,07
50	1024	0	924	1024	2638	11	2,58	13422	33	13,11
60	1024	0	986	1024	3409	9	3,33	21734	40	21,22

A 9. táblázat további oszlopainak fejlécei a korábban említett *rendszerszintű statisztikai fájlban* rögzített jellemzők nevei. A rendszerszintű statisztikai fájl formátuma a napló fájlhoz hasonlóan szöveges, de egyedi és az egyszerű gépi feldolgozáshoz készült. Tabulátorokkal elválasztott számmezőket tartalmaz, a pontosvesszővel kezdődő sorok csak az emberi olvasást segítik. Egy példa a 9. táblázat összeállítása során végzett mérések eredményeként létrejött egyik rendszerszintű statisztikai fájlra:

```
;rounds  repeats  procs   faults  groups  inval  topo   inj
  1024         0    100       60       0      0     1     2
;tfrfg   tfrff    coeff
 0.500   0.500    0.800
;iextra           textra          jextra
     --               --              --

;dextra
     --

;progress indicator: [************************************************]

;incorr  incomp   benign  malign
   1024      0      986    1024
;gtot    gmin     gmax    gavg    gdev
  40651     25       55   39.70    4.70
;ftot    fmin     fmax    favg    fdev
  61749     45       75   60.30    4.70
;mgtot   mgmax    mgavg   mgdev   mgint
   3409      9     3.33    1.75    0.22
;mftot   mfmax    mfavg   mfdev   mfint
  21734     40    21.22    5.48    0.00
;utot    umax     uavg    udev    uint
      0      0      0.0     0.0     0.0
```

Innen kell a megfelelő mezőket pl. egy Excel táblázatba kigyűjteni. Minket többnyire a diagnosztikai minőséget jellemző tévedések száma érdekel: a tévedést tartalmazó diagnosztikai menetek száma (`"incorr"`, `"incomp"`, `"benign"`, `"malign"`), de annál is inkább a jóindulatú (`"mgmax"`, `"mgavg"`) vagy rosszindulatú (`"mfmax"`, `"mfavg"`) tévedéssel félreminősített egységek száma. Ezek mutatják meg, hogy milyen mértékben romlik a diagnosztikai teljesítmény a vizsgált paraméter változásának függvényében. A nevek értelmezését a 10. táblázat adja meg.

A mérés kiértékeléséhez nem elég adatokat gyűjteni, azokat értelmezni is kell és meg is kell jeleníteni. Ennek érdekében az kapott eredményeket szervezzük táblázatokba, ábrázoljuk azokat diagramokon és ennek alapján vonjunk le következtetéseket.

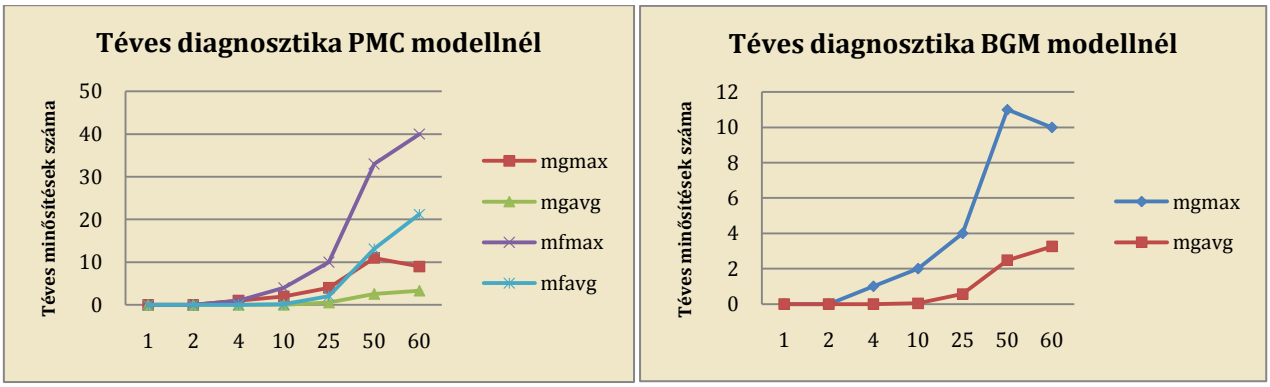
Lássunk erre is egy példát! Az adatgyűjtés egy részét már elvégeztük: a 9. táblázatban látható eredményeket kaptuk a 4.3. fejezetben közölt példa algoritmus futtatásakor 10×10-es tórusz topológiájú rendszerben növekvő hibaszám hatására homogén PMC tesztérvénytelenítés mellett. Ezeket az eredményeket láthatjuk szemléletesen a 9. ábra bal oldalán közölt diagramban. Látható, hogy a t-korlát értékéig (ami a kétdimenziós tórusz topológiánál: 4) szinte egyáltalán nem téved még ez az egyszerű valószínűségi algoritmus sem. (Természetesen egy determinisztikus algoritmusnak a t-korlát értékével *megegyező hibaszámnál sem szabad tévednie*!)

10. táblázat: A rendszerszintű statisztikai fájlban rögzített jellemzők leírása

Jellemző neve	Leírása
"incorr"	Legalább egy diagnosztikai tévedést adó diagnosztikai menetek száma
"incomp"	Nem teljes diagnosztikai képet adó diagnosztikai menetek száma
"benign"	Jóindulatú tévedést tartalmazó diagnosztikai menetek száma
"malign"	Rosszindulatú tévedést tartalmazó diagnosztikai menetek száma
"gtot", "gmin", "gmax", "gavg", "gdev"	A hibátlan egységek összes, minimális, maximális és átlagos száma és szórása
"ftot", "fmin", "fmax", "favg", "fdev"	A hibás egységek összes, minimális, maximális és átlagos száma és szórása
"mgtot", "mgmax", "mgavg", "mgdev", "mgint"	A jóindulatú tévedéssel (jót→rosszra) félreminősített egységek összes, maximális és átlagos száma, szórása + konfidencia intervalluma
"mftot", "mfmax", "mfavg", "mfdev", "mfint"	A rosszindulatú tévedéssel (rosszat→jóra) félreminősített egységek összes, maximális és átlagos száma, szórása + konfidencia intervalluma
"utot", "umax", "uavg", "udev", "uint"	Az ismeretlen ("UNKNOWN" minősítésű) egységek összes, maximális és átlagos száma, szórása + konfidencia intervalluma

Az is látható, hogy mind a jóindulatú, mind a rosszindulatú tévedések száma meredeken emelkedik 25% hibás egység aránynál, majd 50% felett a növekedés üteme csökken. Mi okozhatja ezt? A 25–50% hibaarány tartományban elkezd növekedni annak az esélye, hogy a hibás tesztelő egységek egy tesztelt egység környezetében túlsúlyba jussanak, és így diagnosztikai tévedést okozzanak. Azonban ez a folyamat 50% környékén telítésbe jut, hiszen ha egységre 3 hibás tesztelő jut, akkor azok már tévedésre kényszeríthetik az algoritmust. Ezután már hiába növekszik a hibás tesztelők száma egy egység környezetében 3-ról 4-re a hibaarány növekedésével, az már nem fogja akkora mértékben növelni a diagnosztikai tévedés valószínűségét.

A 9. ábra jobb oldalán közölt diagram ugyanennek a 10 × 10-es tórusz topológiájú rendszernek a viselkedését mutatja szintén növekvő hibaszám hatására, de ezúttal homogén BGM tesztérvénytelenítés mellett. Látható, hogy a rosszindulatú tévedések teljességgel hiányoznak a diagramból. Ez nem meglepő, hiszen a BGM tesztérvénytelenítés sajátosságai miatt ilyen tévedésre nincs lehetőség: a hibás tesztelők a hibás tesztelt egységet csak 1-es teszteredménnyel tesztelhetik, viszont a „rosszat→jóra” típusú tévedéshez a 0-ás teszteredmények túlsúlyára lenne szükség egy hibás tesztelt egység környezetében. Az is látható, hogy a jóindulatú tévedések tekintetében a PMC és a BGM tesztérvénytelenítéssel kapott eredmények lényegében megegyeznek. Ez várható is, hiszen az algoritmus szempontjából a jóindulatú tévedésekre való „hajlam”, azaz e tévedések mechanizmusa egyforma a PMC és a BGM modellek esetén.



9. ábra: A 4.3. fejezetben bemutatott példa algoritmus viselkedése PMC és BGM tesztérvénytelenítés esetén

5 Ellenőrző kérdések

1. Mit nevezünk hibafedésnek? Mikor teljes egy teszt? Van-e különbség a részleges hibafedés és a nem teljes teszt között?
2. Mit jelent a tesztérvénytelenítés? Nevezzen meg két elterjedt tesztérvénytelenítési modellt!
3. Mi az általánosított tesztleképzés? Milyen táblázattal írható le? Mit jelentenek a táblázatban szereplő *c* és *d* szimbolikus konstansok?
4. Mik a PMC tesztérvénytelenítési modell tulajdonságai? Hogyan jelöljük a PMC modellt az általánosított tesztleképzés jelölésrendszere szerint?
5. Mi a tesztelési gráf? Mik a kölcsönös tesztek?Mondjon két olyan gráf topológiát, amit támogat a mérésben használt szimulációs környezet!
6. Mi a detektálhatóság? Mikor *t*-hiba detektálható egy adott rendszer?
7. Mi a diagnosztizálhatóság? Mik az egy lépésben *t*-hiba diagnosztizálhatóság feltételei PMC tesztérvénytelenítési modell esetén?
8. Mi a javítva *t*-hiba diagnosztizálhatóság alapgondolata? Mikor működik ez a megközelítés? Mik az előnyei és a hátrányai?
9. Mi a különbség a determinisztikus és a valószínűségi diagnosztikai algoritmusok között? Mik a valószínűségi algoritmusok előnyei és a hátrányai a determinisztikus algoritmusokhoz képest?
10. Mik a Meyer és Masson féle diagnosztikai algoritmus főbb lépései? Milyen topológia és milyen tesztérvénytelenítés szükséges ahhoz, hogy alkalmazhassuk?