Hibakeresés
párhuzamos és elosztott programokban

Lovas Róbert

BME Automatizálási és Alkalmazott Informatikai Tanszék

2002.

TARTALOMJEGYZÉK

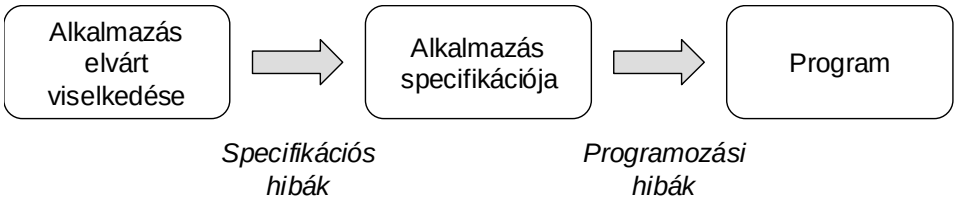
1 Bevezetés

A fejezet első részében a párhuzamos és elosztott programok helyességének elősegítését megcélzó hibakeresési tevékenységet (*correctness debugging*) általánosan vizsgáljuk a programfejlesztés kontextusában, valamint a szekvenciális programok hibakeresésének legfontosabb jellemzőit tekintjük át. A fejezet második részében a párhuzamos és elosztott programok hibakeresésének fontosabb problémaköreit tárgyaljuk.

A segédletben rövidített formában elosztott hibakeresésként (*distributed debugging*) hivatkozunk a párhuzamos és elosztott programok hibakeresésére.

1.1 A hibakeresési tevékenység

Tételezzük fel, hogy adott egy alkalmazás és egy hozzá tartozó specifikáció, ami leírja az alkalmazás elvárt viselkedését. Ideális esetben a specifikáció formális jelölésrendszert használva biztosíthatja, illetve elősegítheti, hogy az alkalmazásunk valamilyen megvalósítása helyes legyen az adott programozási modelltől függően. Abban az esetben, ha az alkalmazás kódját automatikusan tudnánk generálni a specifikációból, akkor helyes (és hatékony) programkódot kapnánk és hibák (*bugs*) csak a specifikáció szintjén jelenhetnének meg (*specification bugs*), ahogy azt az 1. ábra mutatja. Általános és automatikus kódgenerátor eszközök hiányában – amelyek magas szintű specifikációs nyelvekből lennének képesek kódot generálni – a programozó marad a felelős a formális specifikáció programkódra történő leképzéséért. Mindamellett minden programozási modell könnyű érthetőségével és kifejezőképességével próbál hozzájárulni az előbb említett leképzés megkönnyítéséhez, de még akkor is a leképzési tevékenység lehetőséget ad egy másik típusú hiba bevezetésére a programtervezés és megvalósítás során; a programozási hibákra (*programming bugs*), ahogy azt az 1. ábra illusztrálja.



1. ábra - Specifikációs és programozási hibák

Általános gyakorlat, hogy sajnos nem készül el az alkalmazás teljes formális specifikációja, tehát a program helyessége csak az alkalmazás elvárt viselkedéséhez kapcsolható, ami egyedül a fejlesztő fejében létezik. Ez teszi a hibakeresési tevékenységet rendkívül bonyolulttá, hiszen a programozási és

specifikációs hibák ugyanazon a szinten jelentkeznek: beágyazva a programkódba[1].

A fenti nehézségeknek köszönhetően, a hibakeresés alapvetően fontos, bár nem igazán közkedvelt tevékenységgé vált.

Az elmúlt 50 év folyamán hatalmas erőfeszítéseket tettek a szekvenciális programok hibakeresése területén. Számos jelentős hibakeresési technika került kifejlesztésre az alkalmazott programozási absztrakcióktól és paradigmáktól függően (pl. imperatív vagy deklaratív) [16] megcélozva mind a specifikációs, mind a programozási hibákat. Mivel a szekvenciális alkalmazást szekvenciális rendszeren hajtjuk végre, alkalmazható az állapot-alapú megközelítés, hogy megvizsgáljuk az alkalmazásunk viselkedését. A felhasználók számára interaktív hibakeresők teszik lehetővé, hogy a szekvenciális számítás során megvizsgálják az egymás utáni állapotok sikeres végrehajtását. A determinisztikus viselkedésnek köszönhetően könnyű a programot újra és újra végrehajtani (adott bemeneti feltételek mellett), hogy részletesebben megvizsgálhassuk alkalmazásunk viselkedését[2].

A szekvenciális hibakeresés egyszerűségének másik oka, hogy a felhasználónak a hibakeresési folyamat minden egyes lépésében csak egyetlen vezérlési szálról kell gondoskodnia. A szekvenciális programok által esetlegesen létrehozott nagy állapottér könnyen kezelhető a feltételes vagy kódrészletre elhelyezett töréspontok segítségével (ahol a hibakereső eszköznek meg kell állítania a futást). A hibakereső eszköz ilyen kívülről történő vezérlési beavatkozása nincs logikai hatással a szekvenciális számítás viselkedésére.

A párhuzamos és elosztott alkalmazások fejlesztésének jóval nagyobb komplexitása még bonyolultabbá teszi a fenti hibakeresési megközelítést.

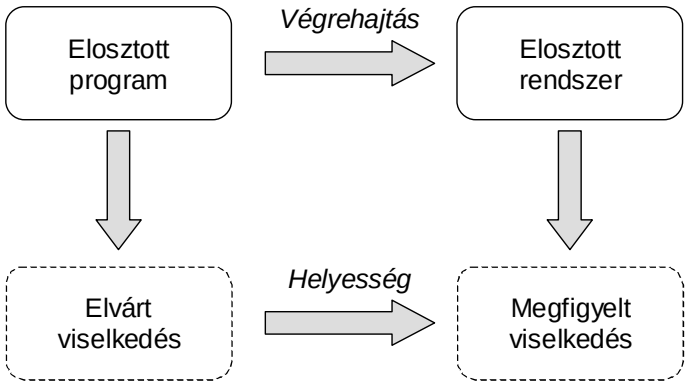
1.2 Elosztott hibakeresés

A fejezetben az *elosztott programot* szekvenciális folyamatok együttesének tekintjük, ahol a folyamatok valamilyen elosztott memóriát feltételező kommunikációs modellt használva tudnak együttműködni. Itt az elosztott kifejezés a hangsúlyos, mivel ezek a folyamatok nem alapozhatnak a fizikailag megosztott (közösen elérhető) memóriára, vagy akár a globális óra absztrakciójára (ami jellemzően szinkronizációs célokat szolgálhat). A párhuzamos alkalmazások is természetesen ebbe a körbe sorolhatóak, amennyiben elosztott memóriás környezetben futnak, mint például egy klaszter (*cluster*) számítási platformon. Az elosztott programok egy adott programozási nyelv szemantikájára alapulnak, amely kifejezi a konkurenciát, a párhuzamosságot, az elosztottságot, valamint a kommunikációt a folyamatok között. Az elosztott programokat elosztott rendszerkörnyezetben hajtjuk végre, és a rendszerkörnyezet biztosítja azokat a számítási mechanizmusokat, amik lehetővé teszik az elosztott program

[1] A programkód szintje alatt még számításba vehetjük azokat a hibákat is, amik az operációsrendszer-hívásokra és a gépi kódra történő leképzés során keletkezhetnek, de általánosságban ezek a hibák az alkalmazásfejlesztő látókörén kívülre esnek.

végrehajtását azon a (virtuális) architektúrán, amit az operációs rendszer és hardver platform együttesen definiál.

Tételezzük fel, hogy kifejlesztettünk egy elosztott programot, aminek helyességéhez kapcsolódó jellemzőket szeretnénk specifikálni, azaz kijelentéseket megadni a programunk elvárt viselkedéséhez, majd egy elosztott hibakeresővel automatikusan összehasonlítani azokat a program megfigyelt viselkedésével (lásd 2. ábra).



2. ábra - Elvárt és megfigyelt viselkedés

A megkövetelt jellemzők specifikálására van mód ugyanúgy, mint a programozási hibákhoz kapcsolódó nemkívánatos, vagy hibás helyzetek detektálására, de ezek megkövetelik az elosztott hibakeresőtől, hogy támogassa az alábbiakat:

- Nyelv, hogy specifikáljuk a helyességhez kapcsolódó kijelentéseinket.
- Algoritmusok a fenti kijelentések kiértékeléséhez.
- Megfigyelési és vezérlési mechanizmusok, hogy a felhasználó képes legyen megvizsgálni a számítás aktuális állapotát, ami a hibás szituációhoz tartozik.

A fent felvázolt ideális képet nehéz megvalósítani egy aszinkron elosztott rendszerben, amely nem rendelkezik sem globális órával, sem globálisan elérhető közös memóriával, sem megkötésekkel az üzenetek átviteli idejével kapcsolatban. Az alábbi felsorolt problémakörök teszik az elosztott hibakeresést jóval nehezebbé, mint a szekvenciális hibakeresést [36]:

1. Nagy számú párhuzamos és elosztott entitás (folyamat/szál), melyek dinamikusan hatnak egymásra az elosztott programban.
2. Az elosztott program eredendően nem-determinisztikus viselkedése.
3. Pontos és konzisztens megfigyelések létrehozása az aszinkron elosztott rendszer globális állapotaira vonatkozólag.
4. A megfigyelési és vezérlési mechanizmusnak során fellépő próba-effektus.

Az első problémakör (elosztott, dinamikusan együttműködő entitások) a nagyszámú számítási állapottal, valamint az előre nem megjósolható dinamikus interakciókkal foglalkozik, melyek befolyásolják a program globális viselkedését. Hogy megoldjuk ezt a problémát, az elosztott hibakeresőnek képesnek kell lennie megfigyelnie a számításokat mind globális szinten (hogy kezelhesse az interakciókat), mind az egyes folyamatok szintjén. Természetesen ezeket a kívánalmakat figyelembe kell venni a hibakeresési eszköz funkcionalitásainak és szoftverarchitektúrájának tervezésekor is.

A második problémakör (nem-determinisztikus viselkedés) miatt az aktuális végrehajtás során tapasztalható viselkedés függ a folyamatok aktuális végrehajtási sebességétől (a különböző processzorsebességeknek és az operációs rendszer különböző ütemezési hatásainak következtében), valamint az előre megjósolhatatlan kommunikációs késleltetésektől is. Egyrészt a nem-determinisztikus viselkedés előnyös, mivel hozzájárulhat a párhuzamos feldolgozás során a várható számítási sebesség növekedéséhez. Másrészt, ez a forrása annak a veszélyes szituációnak, amikor két konkurens akció (két különböző elosztott folyamatból indítva) konfliktusba kerül, és tetszőleges sorrendben hajtódhatnak végre, az előbb már ismertetett időzítési hatások miatt. Az ilyen típusú versenyhelyzetek akár közös memóriás kommunikációs modelleknél, akár üzenetközvetítésen alapuló modelleknél felléphetnek. Ez a problémakör megköveteli az elosztott hibakeresőtől, hogy biztosítson lehetőséget az ilyen típusú helyzetek felismerésére, és úgy értékelje ki a program helyességéhez kapcsolódó kijelentéseket, hogy azoknak érvényesnek kell az összes lehetséges végrehajtási sorrendre. Továbbá szükséges, hogy tegye lehetővé az ilyen hibás helyzetek felhasználói interakciók nélküli ismételt megfigyelését.

A harmadik problémakört (globális állapotok megfigyelése) azért kell számításba vennünk, mivel a hibás szituációk kiértékelése függ magától az akkurátus megfigyeléstől is. Egy elosztott rendszerben ez csak közelítőleg érhető el távoli megfigyeléssel, üzenetközvetítéses alapon, tehát szembe kell nézni azzal a problémával, hogy valójában mi is a rendszer globális állapota. Ezért az elosztott hibakeresőnek biztosítani kell egy megfelelő stratégiát a konzisztens számítási állapotok megfigyelésére.

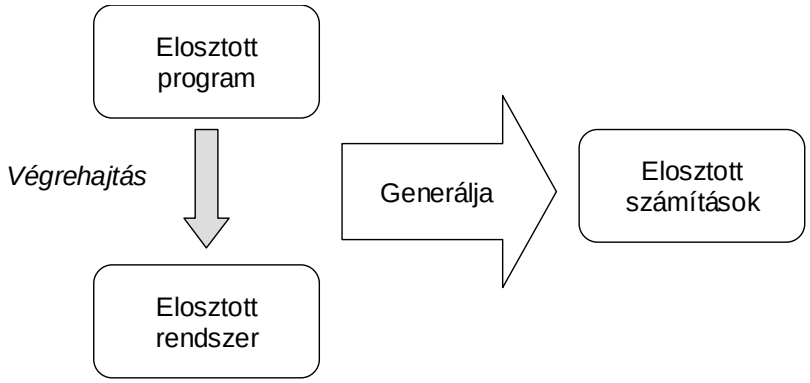
A negyedik problémakör (próba-effektus) azzal az elkerülhetetlen ténnyel állít minket szembe, hogy az elosztott program tanulmányozása során bármely megfigyelés hatással van magára a megfigyelt rendszerre is, tehát az elosztott hibakeresőnek olyan technikán kell alapulnia, ami a lehető legkisebb beavatkozást okozza, ugyanakkor megengedi a felhasználói interakciókat is, még akkor is, ha az elosztott rendszerbe azok rendkívüli mértékű beavatkozást jelentenek.

Azért, hogy megértsük az elosztott hibakeresés hogyan próbálja a fenti problémaköröket megoldani, néhány formális elvet ismertetnünk kell az elosztott rendszerek elméletének területéről.

Az elosztott számítás reprezentálja az *összes* lehetséges viselkedést, amit egy elosztott program egy elosztott rendszeren történő végrehajtása eredményezhet (lásd 3. ábra).

Abból a célból, hogy biztosíthassuk egy elosztott program helyességét, meg kell figyelnünk, és vezérelnünk kell az elosztott számításokat, melyek a program futtatásakor jönnek létre. Így a hibakeresési tevékenység egy még általánosabb

probléma, az elosztott számítások megfigyelésének és vezérlésének részproblémájává válik, mivel a már említett helyességhez kapcsolódó kijelentéseket a számítás minden *jelentősebb* állapotában ki kell értékelnünk.



3. ábra - Elosztott számítások

1.3 A segédlet szerkezete

A segédletnek két fő célja van. Az egyik, hogy megmagyarázzuk annak az okát, miért is olyan nehéz kifejleszteni egy elosztott hibakereső eszközt, ami az eddigiekben felvetett problémakörökre megoldást tud adni. A másik cél az, hogy megtárgyaljuk mindazokat a legfontosabb módszereket és technikákat, melyeket már jelenleg is használnak az elosztott hibakeresők, továbbá azok valószínű továbbfejlődését.

A segédlet többi fejezetében ismertetjük azokat a megoldásokat, amik az elosztott hibakeresési tevékenység tükrében a fent ismertetett akadályok leküzdését célozzák meg. Az elosztott számítás elvét a 2. Fejezetben tárgyaljuk, a megfigyelési problémával a 3. Fejezet foglalkozik. A 4. Fejezet témája a globális kijelentések detektálása. Áttekintést adunk a legfontosabb elosztott hibakeresési metodikákról és technikákról az 5. és 6. Fejezetekben. Végezetül a 7. Fejezetben összefoglaljuk az elosztott hibakereső eszközök megvalósításának legfontosabb problémáit, és rövid összegzés következik a 8. Fejezetben.

A legfontosabb referenciák a [1][2][3][5][43][47] –ben találhatóak.

2 Elosztott számítások

Az ebben a fejezetben leírtak általánosan alkalmazhatóak elosztott programozási modellekre akár folyamatokra, akár szálakra alapul, mind üzenetközvetítést, mind osztott memóriát használó kommunikációs modell esetén, szinkron és aszinkron kommunikációra is.

Egy **elosztott program** általában a konkurencia és kommunikáció absztrakt modelljére alapul. Működési szemantikája megadható eseményekkel, amelyek a folyamat vezérléséhez és a kommunikációs akciókhoz kapcsolódnak. A magas szintű entitások (pl. folyamatok) és események alacsony szintű eseményekre

(primitívekre) képződnek le a program alatt lévő elosztott rendszer segítségével. Bizonyítható, hogy így bármilyen elosztott rendszer megadható folyamatok együttesével, melyek alapvetően üzenetközvetítésen alapuló modellel kommunikálnak egymással (klasszikus küldő és fogadó primitívekkel). Az elosztott rendszerek magukon viselik az aszinkron rendszerek jellegzetességit is, így nincs egy pontos globális fizikai idő referenciánk, hogy a számítási állapotok láncolatát végigkövethessük, ami pl. egy hiba okához vezet az elosztott programban.

Az elosztott rendszerekben fellépő változó folyamat-sebességeknek és különböző üzenet-átviteli késleltetéseknek köszönhetően, tetszőlegesen különböző végrehajtási útvonalak keletkezhetnek, amikor ismételten végrehajtjuk az elosztott programot (adott bemeneti feltételek mellett), így különböző eredményeket szolgáltathat a programunk. Az ilyen típusú, ún. nem-determinisztikus viselkedés teszi igazán bonyolulttá a helyesség, mint tulajdonság kiértékelését, aminek természetesen nemcsak egy megfigyelt útvonal mentén kell fennállnia, hanem az összes lehetséges végrehajtási útvonal mentén.

Az **elosztott számítás** leírja az elosztott program összes lehetséges futását az elosztott rendszeren. Az [5] részletesen bemutatja az elosztott számítások és a globális állapotok megfigyelésének alapvető problémáit. Az egyik alkalmazott koncepció az egyes folyamatok *lokális történetének* elve, a másik az *ok-okozat* kapcsolat, amit a folyamat lokális rendezettsége, valamint a folyamatok közötti interakciókból származó esemény-függőségek határoznak meg [29].

Egy P_i folyamat megadható események sorozatával, melyet lokális történetnek (*local history*) nevezünk (LH_i). Két típusú eseményt különböztetünk meg: *belső események* reprezentálják P_i lokális állapotátmeneteit, melyek nem kapcsolódnak az elosztott rendszer egyetlen másik folyamatához sem, és *interakció események* reprezentálják a folyamatok közti kommunikációt az üzenetküldésre és fogadásra vonatkozóan. P_i lokális történetének teljesen rendezett eseményei reprezentálják P_i változóinak fejlődését és azokat az interakciókat, melyekben P_i is részt vett az elosztott program végrehajtása során.

$$LH_i = e_i(0), e_i(1), \ldots, e_i(f)$$

Ebben a sorozatban $e_i(0)$ a P_i folyamat kezdőeseménye, ami megadja a folyamat kezdőállapotát, $LS_i(0)$-t. Általánosságban a folyamat történetének k-dik eseménye, $e_i(k)$ létrehozza az $LS_i(k)$ lokális állapotot, mint az $e_i(k)$ esemény fellépte utáni közvetlen állapotot. A folyamat végső eseménye $e_i(f)$, és $LS_i(f)$ a folyamat végső állapota[3].

A LH_i lokális történet k-dik eseményével záródó prefix sorozatát $LH_i(k)$-val jelöljük, és P_i részleges történetét reprezentálja a kezdőállapottól egy adott pontig.

A globális történet az összes lokális történet uniójának halmaza:

$$GH = LH_1 \cup LH_2 \cup \ldots \cup LH_N$$

[3] A végtelen folyamatok tárgyalása kívül esik ennek a fejezetnek a keretein, de hasonló következtetések vonhatóak le abban az esetben is.

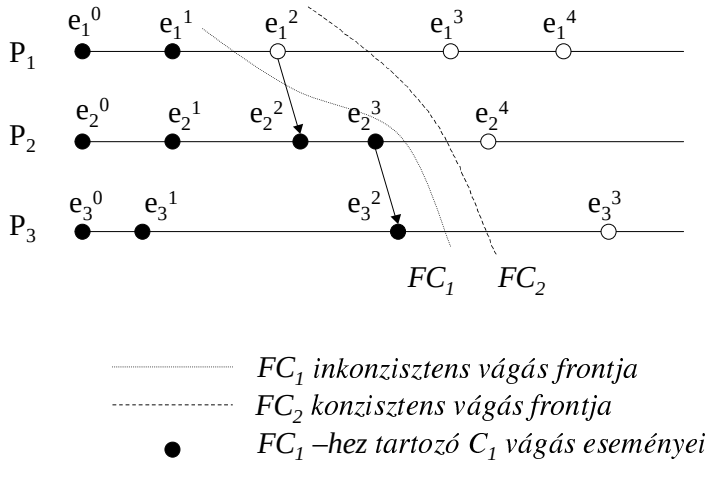
Feltételezhetjük, hogy fix számú (*N*) folyamatunk van, anélkül hogy az általánosságból veszítenénk. Az összes eseménysorrend közül, amit a globális történet reprezentál, csak néhány léphet fel, melyek kielégítik az ok-okozati relációt (→), ahogy azt Lamport definiálta [29]: két esemény között fennáll az $e \rightarrow e'$ reláció, akkor és csak akkor, ha *e* kauzálisan (ok-okozati viszonyban) megelőzi *e'*-t. Illetve fennáll az $e \parallel e'$ reláció, akkor és csak akkor, ha se $e \rightarrow e'$, se $e' \rightarrow e$ nem igazak. Habár ez a reláció csak egy lehetséges okozati függőség, általánosan használt alap az elosztott hibakeresés során, hogy visszakövethessük a hiba okát.

Egy elosztott számítás egy részlegesen rendezett halmaz a globális történettel és a → reláció párosával meghatározva: (*GH*, →). Intuitívan: az események fizikálisan lehetséges összes kombinációja, melyet az elosztott programnak követni kell az elosztott rendszerben történő összes lehetséges végrehajtása során. Az elosztott számítást reprezentálhatja egy folyamat-idő diagram is, ahol az okozati láncolatok helyett egy központosított globális órával rendelkező rendszerben klasszikusan jelöljük az időt (lásd 4. ábra).

Az elosztott rendszerek **megfigyelési problémája** felveti a globális állapotok problémáját az elosztott számítás során. Egy *globális állapot* (*GS*) a lokális állapotok (*LS*) *N*-elemű vektora:

$$GS = (LS_1, LS_2, \ldots, LS_N)$$

ahol LS_i $(i \in \{1,\ldots,N\})$ a P_i folyamat lokális állapota. Az elosztott számítás kezdő globális állapota (*GS(0)*) az összes folyamat lokális kezdőállapotainak segítségével definiálható, azaz $LS_i(0)$: $\forall i \in \{1,\ldots,N\}$ –vel. Az elosztott számítás végső globális állapota, *GS(f)* az összes folyamat lokális végállapotainak segítségével definiálható, azaz $LS_i(f)$: $\forall i \in \{1,\ldots,N\}$ –vel. A probléma a közbenső globális állapotokkal az, hogy nem léphet fel a lokális állapotok összes lehetséges kombinációja az elosztott program valóságos végrehajtása során…



4. ábra – Folyamat-idő diagram

A folyamat-idő diagramhoz kapcsolódóan a *vágás* megadható a globális történet részhalmazaként. A vágás (*cut, C*) az elosztott számítás egy részleges globális történetét reprezentálja, mivel az összes folyamat egy-egy lokális történetének prefixéből áll össze. A C vágás frontja (*frontier, FC*) az egyes lokális történetek prefixének utolsó eseményei. Az *FC* front felfogható az elosztott számítás globális előrehaladásának egy nézeteként (a végrehajtás egy adott pontjáig), mint az utoljára fellépett események. Ezek alapján már könnyen megadhatunk egy jól-definiált és egyedi globális állapotot minden egyes *FC* vágásfronthoz, ami minden egyes folyamat utolsó lokális állapotát veszi.

Az elosztott hibakeresés szempontjából csak a konzisztens vágások (*CC*) jelentősek. Egy *CC* konzisztens vágás balról zárt a $\rightarrow$ relációra, azaz:

$$\forall e,e' \in GH : e \in CC \wedge e' \rightarrow e \Rightarrow e' \in CC$$

Intuitívan: egy konzisztens vágás magában foglalja összes eseményének a múltját is. Egy vágás, ami tartalmaz olyan *e* eseményt, aminek nem mindent *e*-t kauzálisan megelőző eseménye szerepel a vágásban, akkor az nem tekinthető az elosztott program végrehajtásának egy nézetének.

A konzisztens globális állapotot megkaphatjuk egy *FC* konzisztens vágás vágási frontjával meghatározott globális állapotként. A 4. ábrán az FC_2-höz tartozó globális állapot konzisztens, míg az FC_1-hez tartozó nem.

A konzisztens vágás és konzisztens globális állapot koncepciója felhasználható az elosztott hibakeresést szolgáló megfigyelési modell meghatározásához. Szabadon megfogalmazva; egy elosztott számítás aktuális állapotát megkaphatjuk, ha vesszük a konzisztens vágástól „balra” lévő eseményeket (és állapotokat), ami megegyezik a múlttal, míg a jobbra lévő események a jövőt adják. Ez a megközelítés sugallja, hogy az elosztott számítást inkrementálisan hajtsuk végre az elosztott hibakereső vezérlésével, ahol az egymást követő konzisztens globális állapotokat vizsgálnánk meg, hogy kielégítik-e a helyességhez kapcsolódó kijelentések. Habár ez igen fontos kutatási irány az elosztott hibakereső eszközök területén, ennek kivitelezése számos nehézségbe ütközik, ahogy azt a későbbiekben ismertetjük.

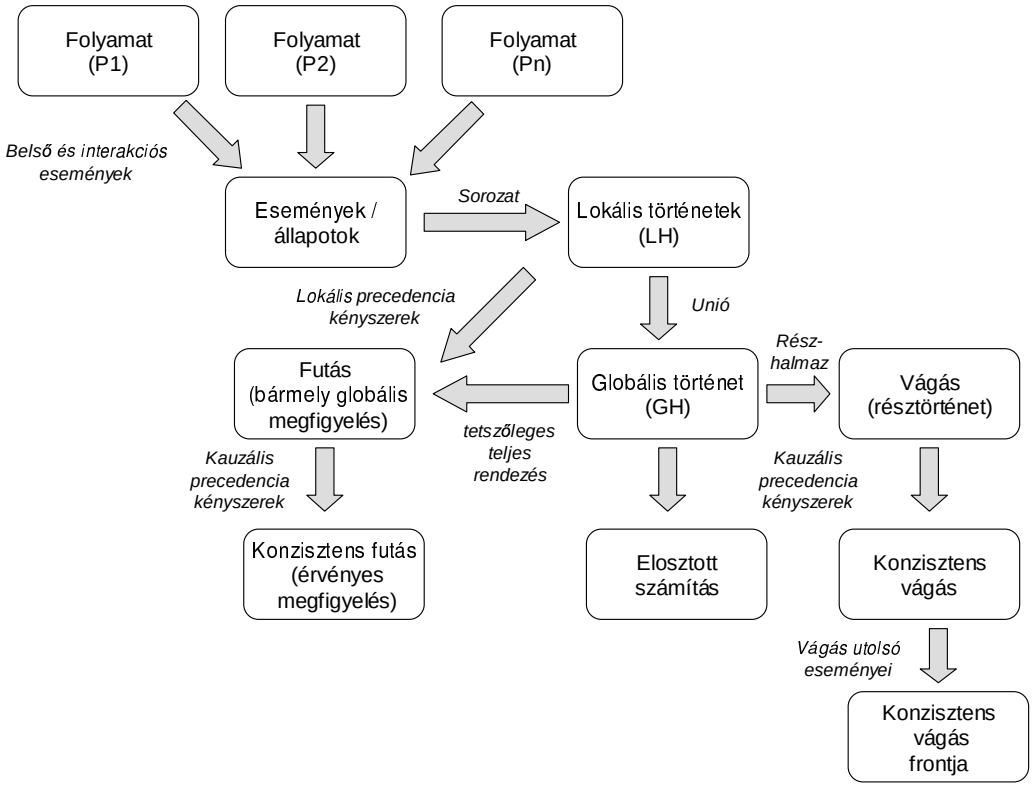
Azért, hogy az elosztott program viselkedését megértsük, számításba kell venni az összes közbenső konzisztens globális állapotot is, ami felléphet a *GS(0)* kezdőállapot és a *GS(f)* végállapot között. Az elosztott program minden egyes végrehajtása során, különböző konzisztens globális állapothalmazt követhetünk végig, tehát minden végrehajtás különböző állapotsorrendet eredményezhet az elosztott rendszer nem-determinisztikus tulajdonsága miatt. A helyesség biztosítása érdekében az összes lehetséges konzisztens globális állapotot meg kell vizsgálnunk.

A konzisztens futás (*consistent run*, *CR*) koncepciója az elosztott számítás egy lehetséges megfigyelésére alapul, amikor az összes esemény teljesen rendezetten jelentkezik, kibővítve a részleges rendezettséget, amit *Lamport* kauzális relációja biztosított. Egy konzisztens futást megkaphatunk a folyamat-idő diagrammból egy eseménysor felépítésével, ami figyelembe veszi az ok-okozati

láncolatot és a konkurens események sorrendjét is. Egy konzisztens futás meghatároz egy konzisztens globális állapot sorozatot:

$$CR = GS_1, GS_2,\ldots,GS_k, GS_{k+1}, \ldots, GS_m$$

Ahol $GS_1 = GS(0)$, $GS_m = GS(f)$ és GS_{k+1} csak egy folyamatnak csak egy lokális állapotában különbözhet GS_k-tól. Szabadon fogalmazva, egy konzisztens futás meghatároz egy konzisztens globális állapotokból álló sorozatot, ahol minden egyes új globális állapotot megkaphatunk egy folyamat „szimpla léptetésével". Bevezethetjük, hogy GS_k *vezet (lead-to)* GS_{k+1}-be egy *CR* konzisztens futás során, ha azok a konzisztens futásban közvetlen szomszédok és *GS* elérhető *GS'*-ből egy *CR* konzisztens futás során, akkor és csak akkor, ha *GS'* |→ *GS* a *CR*-ben, ahol |→ a *vezet* reláció tranzitív lezárása.



5. ábra - Az elosztott számítás elvének összegzése

Habár a konzisztens futás koncepciója mesterségesen megszorítja az elosztott számítást, hiszen egy időpillanatban csak egy esemény történhet, de tetszőleges eseményrendezése reprezentálja az elosztott programvégrehajtás nem-determinisztikusságát. Ha az elosztott számításhoz létre akarjuk hozni az összes lehetséges konzisztens globális állapotból álló sorozatot, figyelembe kell venni az összes lehetséges konzisztens futás halmazát. Az összes konzisztens globális állapot halmaza a *vezet* relációval rendezve meghatározza a globális állapotok *L* rácsát (*lattice*), ami jellemzi az összes lehetséges végrehajtási útvonalat egy adott

elosztott számításnál. *GS(0)* az *infimum* és *GS(f)* a *szuprémum* elemei *L*-nek, és az összes konzisztens futás halmaza az összes *GS(0)* és *GS(f)* közötti útvonal halmaza.

Összegezve, az elosztott program működési szemantikája leírható az elosztott számítások segítségével, amit a program elosztott rendszeren való futásakor kapunk (lásd 5. ábra, és [5] a részletesebb ismertetésért). Az elosztott számítás leírja az összes lehetséges konzisztens globális állapotból álló sorozatot, ami *GS(0)* kezdőállapotból indul és *GS(f)* végállapotba vezet. Az ilyen jellegű lehetséges sorozatokat egyedül a lokális folyamatok eseménysorrendjével és a *Lamport*-féle ok-okozati relációval szorítjuk meg.

Az elosztott program helyességhez kapcsolodó tulajdonságainak ellenőrzése és detektálása szempontjából a fenti útvonalak kimerítő bejárására lenne szükség. A megvizsgálandó konzisztens globális állapotok nagy kombinációja miatt ez a megközelítés általánosságban nem kivitelezhető. Hovatovább már önmagában problémát jelent a konzisztens globális állapotok konstruálása is.

3 Globális állapotok megfigyelése

Szabadon megfogalmazva az elosztott számítás globális állapota nem más, mint lokális állapotok gyűjteménye, amit valamilyen ideális külső megfigyelő láthat. Meg kell jegyezni, hogy a globális állapot, ahogy fent leírtuk, implicit tartalmazza a folyamatok közti kommunikációs csatornák állapotát is. Egy elosztott rendszerben az egyetlen lehetséges mód, hogy egy külső megfigyelő egy ilyen nézetet létrehozhasson, az üzenetváltás minden egyes távoli folyamattal. Ezért számos probléma vetődik fel a megfigyelési probléma kapcsán:

- Egy globális állapot elévültté válhat addigra, amire a külső megfigyelő az aktuális globális nézetet létrehozza. Ez abban az esetben lép fel, ha a megfigyelés on-line történik az elosztott program aktuális végrehajtása során[4]. Célunk lehet az elosztott hibakeresés során egy kitüntetett globális állapot nyomon követése a számítás on-line történő megfigyelésekor, hogy detektálhassuk a hibát. Ehhez szükséges egy mechanizmus, ami képes megállítani az összes folyamat végrehajtását és visszaállítani a lokális állapotokat, amiket így részletesebben is megvizsgálhatunk (a fejezetben később ismertetjük ezeket a technikákat). Ha off-line megfigyelésről van szó, akkor ez a probléma nem jelentkezik, mivel utólag, a program végrehajtása után (*postmortem*) vizsgáljuk meg az elosztott számítás globális történetét.
- A megfigyelt globális állapot az elosztott számítás egy vágása. On-line megfigyeléskor a létrehozott globális állapot csak az elosztott számítás ezen vágására vonatkozik, tehát csak azt tudjuk megvizsgálni, mi történt az adott pontig. Off-line megfigyeléskor mind a múlt, mind a jövő ismert és elérhető a megfigyelő által.

[4] Ez az aspektus rendkívül fontos az elosztott reaktív alkalmazásoknál [5][9].

- A megfigyelt globális állapotnak az elosztott számítás egy konzisztens vágásának kell lennie, de az elosztott rendszerben inkonzisztens állapotok megfigyelésére is sor kerülhet a nem megjósolható üzenetküldési sorrendnek köszönhetően. A megfigyelő könnyen inkonzisztens eseménysorokat építhet fel, amik nem őrzik meg a kauzális precedencia kapcsolatot, aminek megoldását szintén tárgyaljuk a fejezetben.
- Több független megfigyelő különböző nézetet építhet fel ugyanarról az elosztott számításról. Még ha konzisztens vágások biztosítottak is, több független megfigyelő tetszőleges konzisztens vágásokat építhet. Több konkurens és független megfigyelő esetén egy elosztott számításhoz az egységes nézet garantálása összehangolást igényel, és az elosztott rendszerek kutatása foglalkozik vele [5]. Ez egy olyan probléma, amit a legtöbb létező elosztott hibakereső nem támogat, habár az integrált fejlesztőkörnyezetek megjelenésével ennek fontossága egyre növekszik, hiszen a környezet különböző konkurens eszközei megfigyelik (vagy akár vezérlik is) az elosztott számítást. Az eszközök koordinációját röviden megemlítjük a 7. fejezetben is és illusztrálásra került az [55] második részében is.

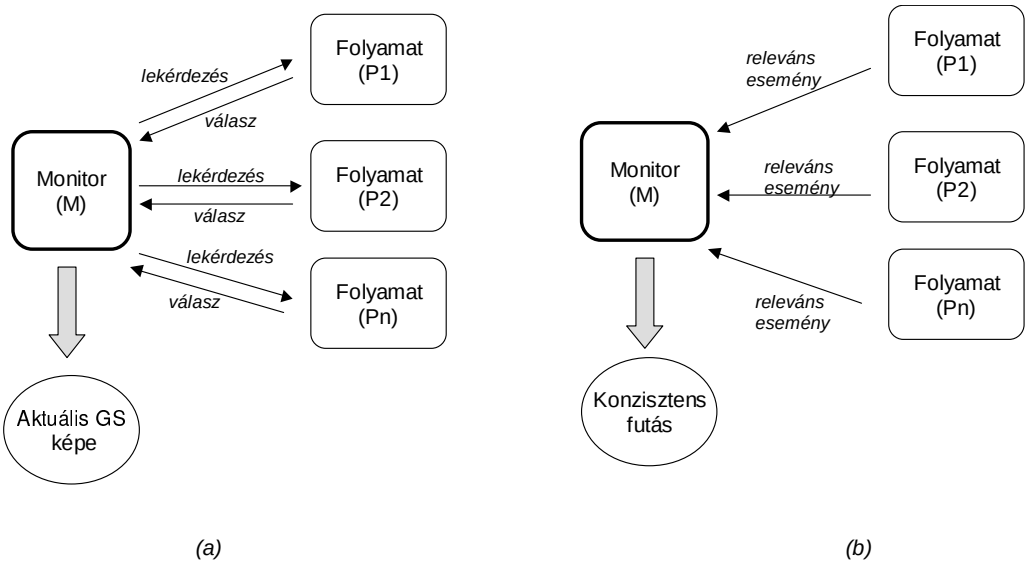
Az elosztott számítások megfigyelésének gyakorlati módszerei nagymértékben függnek az alkalmazott elosztott hibakeresési megközelítéstől.

- *Off-line*: Ebben a megközelítésben az elosztott számítás globális történetét analizálhatjuk, amit az elosztott program előző végrehajtása során generáltunk (vagy akár elosztott program modelljének szimulációjakor). A módszer mindig a teljes történettel foglalkozik.
- *On-line*: Ebben a megközelítésben különböző algoritmusokat kell kifejleszteni, hogy létrehozhassuk a globális állapotot vagy az elosztott számítás egy konzisztens futását. A módszerek a részleges történettel foglalkozik.

Az on-line megfigyelés megvalósításának legfontosabb megközelítései a külső megfigyelő vagy monitor folyamatra alapoznak (M). Az elosztott hibakeresés során ezt használhatjuk, hogy felépítsük az elosztott számítás globális állapotainak pillanatképeit, hogy helyességi tulajdonságokat kiértékelhessük azokon a pontokon. Továbbá felhasználhatóak arra, hogy felépítsük a konzisztens futások megfigyeléseit, melyeket az elosztott program végrehajtása során követ. Minden megfigyelési stratégia esetén foglalkoznunk kell a próba-effekt problémájával, de most feltételezzük, hogy a próba-effektnek elhanyagolható hatása van abban az értelemben, hogy az M és P_i folyamatok közti üzenetváltások nincsenek hatással az elosztott számítás eseményeinek sorrendjére (összevetve a nem monitorozott számítással). A próba-effekt részletes leírása a [55] 6. fejezetében található.

Az on-line megfigyelési megközelítések mindegyike az üzenetközvetítési szabályokkal kapcsolatban számos feltételezésre alapul, amit rá kell kényszeríteni az elosztott rendszerre, kezdve az üzenetek FIFO rendezésére a folyamatpárok között, egészen az üzenetek kauzális kézbesítéséig. Az ilyen feltevések fontosak, hogy a külső megfigyelő konzisztens globális állapotokat építhessen. A kézbesítési

szabályok (pl. vektor dátumbélyegek alapján) részletes leírása túlmutat a segédlet keretein. Itt egy dolgot kell megjegyezni, a vektor dátumbélyegek támogatják azokat az eseménykapcsolatokat, amelyek biztosítják a kauzalitást. Komplett áttekintő tanulmányok a [13][34][46]-ban találhatóak a témával kapcsolatban.



6. ábra - Két megfigyelési megközelítés: (a) Globális pillanatkép (b) aktív értesítés

Az első stratégia megpróbál egy globális pillanatképet létrehozni [7] az elosztott számítás egy konzisztens globális állapotához. Az *M* megfigyelő felelős az összes folyamat lekérdezéséért, és azoknak válaszul el kell küldeniük az aktuális lokális állapotokat (lásd 6. ábra (a)). A megfigyelő ekkor létrehozza a konzisztens globális állapotot, tehát a módszer biztosítja, hogy *L*-ben van egy útvonal, ami ezt a konzisztens globális állapotot tartalmazza. *M* egymást követő lekérdezései használhatóak arra, hogy a konzisztens globális állapotok fejlődését megpróbálhassa felépíteni. Sajnos ez a módszer kihagyhat fontos konzisztens globális állapotokat, vagy akár elfogadhatatlanul nagy költsége is lehet a folytonos lekérdezéseknek köszönhetően, valamint hatalmas késleltetések is felléphetnek a megfigyelésben.

A második stratégia lehetővé teszi *M*-nek, hogy felépítse a teljes megfigyelést vagy az elosztott számítás konzisztens futását (lásd 6. ábra (b)). Minden egyes folyamat felelős azért, hogy értesítse *M*-et, ha valamilyen releváns[5] esemény következik be. *M* begyűjti az összes információt az összes folyamattól, és biztosítja a konzisztens futás felépítését. Az ilyen konzisztens futás kapcsolódik a lehetséges útvonalak közül egyhez (*L-ben*), ami az elosztott számítás aktuálisan követni tudott.

Mindegyik megfigyelési módszernek vannak előnyei és hátrányai is, amit a későbbiek folyamán tárgyalunk meg.

[5] A „releváns esemény” függ attól, hogy milyen típusú állapotváltozásokat akarunk megfigyelni.

4 Globális kijelentések detektálása

Egy általános módszer az elosztott hibakeresés támogatására (számos szerző munkájára alapozva [9][19][25][51]), a következő három lépésből áll: globális helyességi feltételek specifikálása, azok (off-line vagy on-line) kiértékelése, és egy megfelelő reakció az elosztott hibakeresőtől a kiértékelés eredményétől függően [4]. Egy teljes megvalósításnak lehetővé kell tennie, hogy:

1. … specifikálhassuk a viselkedést. Számos megközelítés van, hogyan specifikáljuk a lokális és globális állapotokra vonatkozóan a globális kijelentéseket.
2. … a globális kijelentéseket detektálhassuk. Számos javaslat van az off-line és on-line globális kijelentés detektor eszközre, melyek a globális kijelentések tetszőleges típusát támogatják [6][17].
3. … az elosztott hibakereső reagálhasson. Az elosztott hibakereső eszköznek el kell végeznie a konzisztens globális állapot (off-line vagy on-line) rekonstrukcióját, ami a detektált globális kijelentéshez tartozik.

Ezek a keretrendszer elméleti alapjának problémái, ami megfigyelésre és vezérlésre alkalmas elosztott hibakereső technikák kifejlesztését követelte meg (ld. 6. fejezet).

4.1 Globális kijelentések specifikálása

Ez a lépés az elosztott program kívánatos és nemkívánatos jellemzőinek meghatározásával kezdődik, melyek a program helyességi kritériumaihoz tartoznak.

A jellemzőket, mint globális kijelentések (*global prediction, GP*) fejezzük ki, melyek *boolean* kifejezések különböző feltételeket tartalmazva a folyamatok lokális változóira és a kommunikációs csatornák állapotaira vonatkozóan. Több szerző különböző specifikációs nyelveket javasolt a globális kijelentésekhez, a globális állapotokra vagy eseményekre, illetőleg az állapotok sorozatára, vagy az események mintázatára vonatkozóan [6][17].

Egy ilyen specifikációs nyelv tervezése során jelentkező legfontosabb problémák a következők:

1. A nyelv kifejezőképessége, hogy alkalmas legyen a megkövetelt feltételek specifikálására, a helyességgel vagy a hibás helyzetekkel kapcsolatban.
2. Számítási komplexitás, hogy hatékonyan lehessen implementálni. Egy rendkívül kifejező jelölésrendszer is elveszíti a gyakorlati jelentőségét, ha az NP-teljes kiértékelésre alapul.
3. Elfogadható kompromisszum az elosztott program modelljének absztrakciós szintje és az elosztott hibakereső eszköz megfigyelési szintje között. Magyarul könnyűnek kell lennie a megfeleltetésnek a helyességhez kapcsolódó feltételek, az elosztott program koncepciója (és maga a program kódja), valamint az elosztott

hibakereső eszköz által aktuálisan megfigyelt entitás (pl. az alacsony szintű események vagy globális állapotok) között.

Az elosztott hibakeresés során történő kijelentés-alapú megközelítés alkalmazásának legnagyobb akadálya a globális feltételek kiértékelésének komplexitása.

4.2 Globális kijelentések kiértékelése

Ez a lépés felelős a globális kijelentések detektálásáért, akár off-line, akár on-line megközelítést alkalmazunk. A globális kijelentések általános formájának kiértékelése NP-nehéz problémának bizonyult [8], ezért számos szerző globális kijelentések megszorított formáira fókuszált, úgy mint konjunktív [18][24] és diszjunktív globális kijelentések. Habár a globális kijelentések kötötté váltak, még mindig hasznosak az elosztott hibakeresés során.

Egy fontos különbséget állapíthatunk meg az elosztott programok ún. *stabil* és *instabil* tulajdonságai között. Ha egyszer egy stabil tulajdonság igazzá válik egy adott *GS* globális állapotban, akkor az igaz is marad az összes következő állapotban, ami *GS* állapotból elérhető. A holtpont és a terminálás két jellemző példa a stabil jellemzőkre. Másrészt egy instabil jellemző dinamikusan változtathatja értékét az elosztott végrehajtás során. Mindkét típusú tulajdonság fontos jelentőséggel bír az elosztott programok helyességének vizsgálatakor. Az instabil tulajdonságok az elosztott program dinamikus viselkedésének eredményeként megszülető szituációkra reflektálhatnak, így pl. az elosztott kölcsönös kizárásra.

A stabil és instabil tulajdonságok kiértékelése különböző követelményeket támaszt. A stabil tulajdonságokat megkaphatjuk az on-line megfigyelések során (a globális pillanatkép megközelítése alapján), ha biztosítjuk a lekérdezések ismétlését mindaddig, amíg egy konzisztens globális állapotban a program ki nem elégíti az adott stabil tulajdonságot. Sajnos az instabil tulajdonságok detektálása jóval nehezebb. Az on-line megfigyelések (globális pillanatkép megközelítést alkalmazva) nem biztosítják ezt, mivel a módszer az elosztott számítás során kihagyhat olyan globális állapotokat, ahol a tulajdonság átmenetileg fennállt, köszönhetően az elosztott rendszer bizonytalanságának. Az aktív értesítések alapján történő megfigyeléskor (lásd 6. ábra) jobb a helyzet, de ha a tulajdonság még fenn is áll egy bizonyos globális állapotban a felépített konzisztens futás során, ez nem ad információt arról, a tulajdonság hogyan viselkedik más lehetséges konzisztens futások során.

A globális tulajdonságok kibővített formáit javasolta számos szerző, ami megpróbálja kifejezni az elosztott program viselkedését az egész elosztott számításra vonatkozóan, egyetlen konzisztens globális állapot helyett. Ezeknek az alábbi alakjai lehetnek:

Definitely(GP): az elosztott számítás *összes* konzisztens futásában létezik legalább egy állapot, ahol a globális állapot kielégíti GP-t.

Possibly(GP): az elosztott számítás *valamely* konzisztens futásában létezik egy olyan állapot, ahol a globális állapot kielégíti GP-t.

A második forma kifejezetten hasznos az elosztott hibakeresésnél, hiszen egy tulajdonsághoz, ami kifejez egy nem kívánatos vagy hibás feltételt, elégséges, ha egy olyan konzisztens globális állapotot találunk, ahol az adott *GP* igaz. Ez megköveteli, hogy végigkeressük a konzisztens globális állapotok *L* rácsát, de csak addig, amíg ilyen konzisztens globális állapotot nem találtunk.

Egy helyességi tulajdonság, ami olyan feltételt fejez ki, aminek az elosztott program összes lehetséges futása során fenn kell állnia, akkor az adott *GP*-nek igaznak kell lennie az *L* rács összes útvonalán.

Számos szerző mutatott hasznos megoldásokat, hogy felépítsük és bejárjuk a konzisztens globális állapotok teljes rácsát [9], amelyek alkalmasak mind stabil, mind instabil kijelentések kiértékelésére, *Possibly* és *Definitely* kifejezéseket használva. Más szerzők megpróbáltak bevezetni speciális és egyszerűsített *GP* formákat, mint például a konjunktív vagy diszjunktív formák, hogy elkerüljék az *L* rács kimerítő keresését. Ezen megközelítések összegzése megtalálható a [17]-ben.

Ugyanezekre a célokra egy másik megközelítés a versenyhelyzetek speciális formáival és azok detektálásával foglalkozik. Ezek a mechanizmusok általában, mint implicit funkciók kerültek beágyazásra az elosztott hibakereső eszközökbe [23][41].

4.3 Reakció a globális kijelentések detektálásakor

A kiértékelt logikai kondíciók felhasználói értelmezésének függvényében bizonyos reakciók szükségesek. Például, ha a detektált globális kijelentés egy hibás szituációhoz tartozik, akkor az elosztott hibakereső eszköznek végre kell tudnia hajtania a következő akciókat:

1. Az elosztott program végrehajtásának megállítása.
2. Az összes folyamat lokális állapotának visszaállítása, hogy abba a konzisztens globális állapotba kerüljünk, ami a detektált globális kijelentéshez tartozott.
3. Lehetővé tenni a felhasználó számára, hogy az egyes folyamatok lokális állapotát (az adott konzisztens globális állapotban) és a folyamatok múltját megvizsgálhassa.

Az első akció természetesen csak akkor szükséges, amikor a kijelentések detektálása on-line módon történik. Ennek megvalósítása nyitott a különböző értelmezések miatt (a független számító folyamatok aszinkron előrehaladása miatt). A fenti akciók egyszerűbbek a végrehajtás utáni megközelítés alkalmazása esetén, mivel a számítás teljes története elérhető, sőt még azt is lehetővé teszi a felhasználó számára, hogy a rekonstruált globális állapot előtti és utáni végrehajtási útvonalat is bejárja. A 6. Fejezetben a létező megoldási technikákat tekintjük át.

5 Hibakeresési módszerek

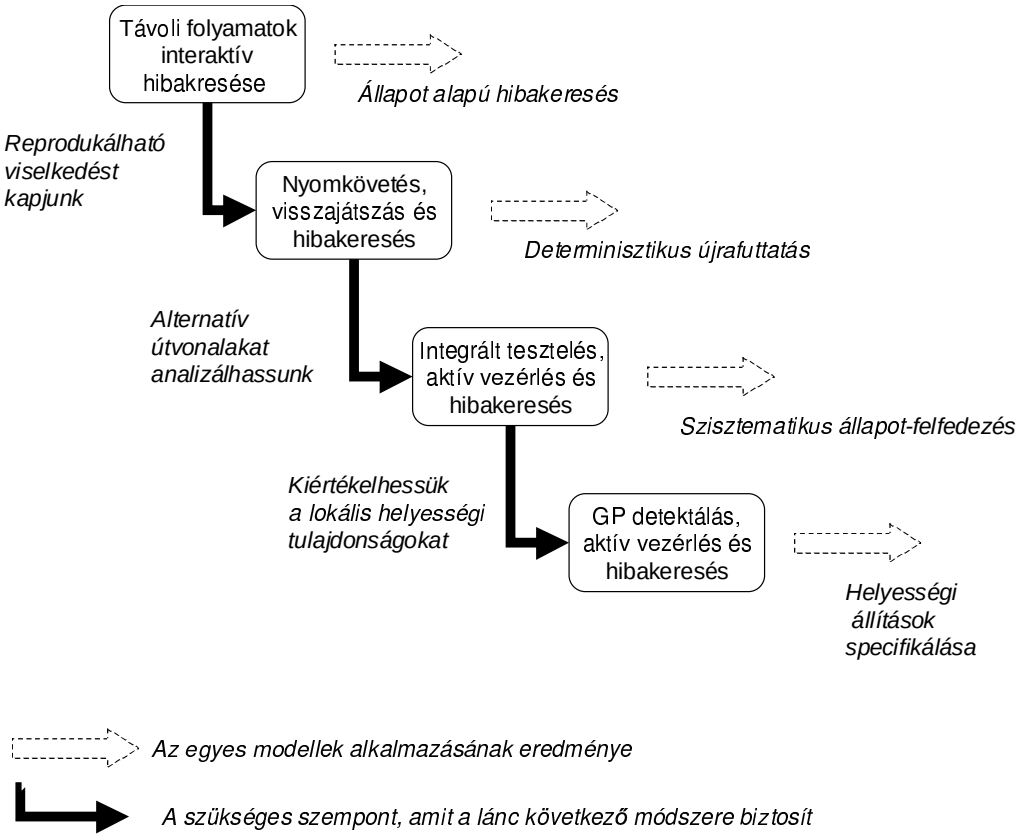
A fejezetben három különböző kritériumot használunk, hogy az elosztott hibakeresési módszereket osztályozzuk: (i) Az elosztott hibakeresési tevékenység mely lépéseit támogatja a fejlesztési ciklusban? (ii) Mikor történik ez a

hibakeresési tevékenység a fejlesztési ciklusban? (iii) Milyen megfigyelési módszert használunk?

5.1 Az elosztott hibakeresési ciklus lépései

Az elosztott hibakeresési módszereket aszerint osztályozhatjuk, hogy milyen szinten támogatják a felhasználót tekintettel a globális kijelentések specifikálása és detektálására, valamint az elosztott program hibáit okozó tényezők felderítésére (lásd 7. ábra).

A következőkben ezeket a megközelítéseket egymás után tárgyaljuk, (kezdve a legegyszerűbbtől az egyre összetettebbek felé), melyek egymás kiegészítői abban az értelemben, hogy minden egyes megközelítés megpróbálja a sorrendben előző határait túllépni.



7. ábra - Elosztott hibakerési módszerek

5.1.1 Távoli szekvenciális folyamatok interaktív hibakeresése

Ez a módszer a hagyományos szekvenciális hibakeresési parancsok kiterjesztésén alapul, amely lehetővé teszi a távoli szekvenciális folyamatok végrehajtásának különálló on-line megfigyelését és vezérlését. Ez egy igen

megszorított megközelítés, mivel az elosztott program különálló folyamatainak lokális történetét engedi csak megvizsgálni. Ahogy minden egyes lokális történet csupán az egyes folyamat előrehaladását írja le (belső és interakciós eseményekkel), a programozó feladata, hogy felépítse a globális képet az elosztott számításról. Habár az ilyen alapvető távoli hibakeresési mechanizmusok megkövetelik, hogy sokkal kifinomultabb módszerek alkalmazását is megengedjük, de a legtöbb kereskedelmi és akadémiai elosztott hibakereső eszköz támogatja. A legnagyobb különbség a létező elosztott hibakereső eszközök között a funkcionalitások és architektúrájuk tervezése között van.

Lényegében ez a megközelítés nem kezeli a nem-determinisztikus viselkedést, amit egy elosztott program produkál egy elosztott rendszeren.

5.1.2 Nyomkövetés, visszajátszás és hibakeresés

Azért, hogy megoldjuk a reprodukálhatóság problémáját, ez a megközelítés az elosztott számítás által generált releváns események nyomkövetési állományainak (*trace file*) összegyűjtésén alapul. A nyomkövetési állomány leír egy számítási útvonalat (konzisztens futást), ami a végrehajtás után analizálható. Ha hibás szituációt találunk, az elosztott programot újrafuttathatjuk egy felügyelő mechanizmus vezérlése alatt, ami a nyomkövetett események sorozatát használja fel arra, hogy rákényszerítse az elosztott számítást, hogy ugyanazt a számítási útvonalat kövesse végig, mint az előző futtatás alkalmával. Ez lehetővé teszi a felhasználó számára, hogy reprodukálható módon ciklikus interaktív hibakeresést alkalmazva megvizsgálja ennek az útvonalnak a viselkedését. Közben a felhasználó támaszkodhat az előző megközelítés által nyújtott megfigyelési és vezérlési funkcionalitásokra. A nyomkövetési és visszajátszási technika intenzív kutatási terület volt az elmúlt évtizedben, leginkább a próba-effekt és a nyomkövetett információ méretének csökkentésével foglalkoztak (lásd 6. fejezet). Sajnos nem mindegyik kereskedelmi hibakereső eszköz támogatja ezt a szolgáltatást. Ez a megközelítés használatos még a rendszerek megfigyelésére (monitoring) is, ahogy azt a [55] 6. fejezete is tárgyalja.

Az elosztott hibakeresés szempontjából egy erős megkötés, hogy ez a megközelítés nem ad támogatást az elosztott számítás nyomkövetett útvonala mellett más útvonalak vizsgálatára. Ha az első futás, ami a nyomkövetési állomány elkészítésére szolgált, egy „szabad” futás (azaz felügyelő mechanizmussal által nem vezérelt) volt, az eredményül kapott nyomkövetési fájl csak egy véletlenszerű útvonal a nagyszámú lehetséges útvonal közül. Ez nem garantálja, hogy az aktuális útvonal érdekes vizsgálati szempontból.

5.1.3 Integrált tesztelés, aktív vezérlés és hibakeresés

Ez a megközelítés megpróbál túllépni az fent említett egyszerű passzív nyomkövetés és visszajátszás korlátjain. Több szerző javasolt különböző megközelítéseket az elosztott programok végrehajtásának aktív vezérlésére elosztott hibakereséshez. Mindegyiknek hasonló célja van; szolgáltatást nyújtson arra, hogy a végrehajtást az elosztott számítás egy útvonalára kényszerítse, így megkönnyítve a hibás szituációk lokalizálását (de a módszerben különböznek, hogy miképp generálják azt a szükséges konzisztens futást, amit a vezérelt

végrehajtásnak követnie kell). A következőkben az egyik ilyen megközelítést részletezzük az illusztráció kedvéért.

A megközelítés két különálló fázisban tárgyalja az elosztott hibakeresést: a statikus analízis és a tesztelési fázis (T fázis), valamint a dinamikus analízis és a hibakeresési fázis (D fázis) integrálásán alapul. A T fázis célja, hogy segítsen a felhasználónak az olyan érdekes konzisztens futások generálásában, ahol a helyességi tulajdonságok sérülhetnek. Általánosságban nem kivitelezhető (nem is lehetséges), hogy teljesen automatikusan történjen a T fázis, de egy interaktív tesztelési eszköz hasznos lehet, hogy együttműködve a felhasználóval meghatározzák azokat a feltételeket és az elosztott program azon kódrégióit, ami az analízis tárgyát fogják képezni. A T fázist ezután arra használjuk, hogy a parancsok egy olyan sorozatát generáljuk, amit az elosztott program futásának vezérlésére használható. Egy ilyen elosztott programfuttatás pedig már tárgya lehet a nyomkövetés és visszajátszás megközelítésnek, és integrálva lehet egy ciklikus hibakeresésbe.

A legfontosabb előnye ennek a módszernek, hogy lehetővé teszi a felhasználó számára, hogy interaktívan „átsétáljon” a T és D fázisok között, amíg nem győződött meg a helyességi kritériumok fennállásáról. Egy másik előnye ennek a megközelítésnek, hogy ötvözi a statikus és dinamikus analízis előnyeit, hogy segítsen a felhasználónak megértenie az elosztott program viselkedését.

A legfőbb probléma ezzel a módszerrel, hogy a felhasználó meggyőződésére alapszik, hogy az összes releváns forgatókönyv specifikálva, generálva, tesztelve és analizálva lett, és így bízhat csak az elosztott program helyességében. Nincs teljes garancia, hogy nem lett fontos szituáció kihagyva.

5.1.4 Globális kijelentések automatikus detektálása, aktív vezérlés és hibakeresés

Ez a megközelítés kísérletet tesz arra, hogy a felhasználó bizalmát növelje az előző megközelítés eredményeinek kapcsán, megengedve a helyességi kritériumok specifikálását globális kijelentésekkel. Az ilyen globális kijelentéseket aztán automatikusan kiértékeli a detektáló algoritmus (akár off-line, akár on-line üzemmódban). Ennek a megközelítésnek a legfontosabb céljait az előző fejezetben már megadtuk.

5.2 Elosztott hibakeresés időzítése

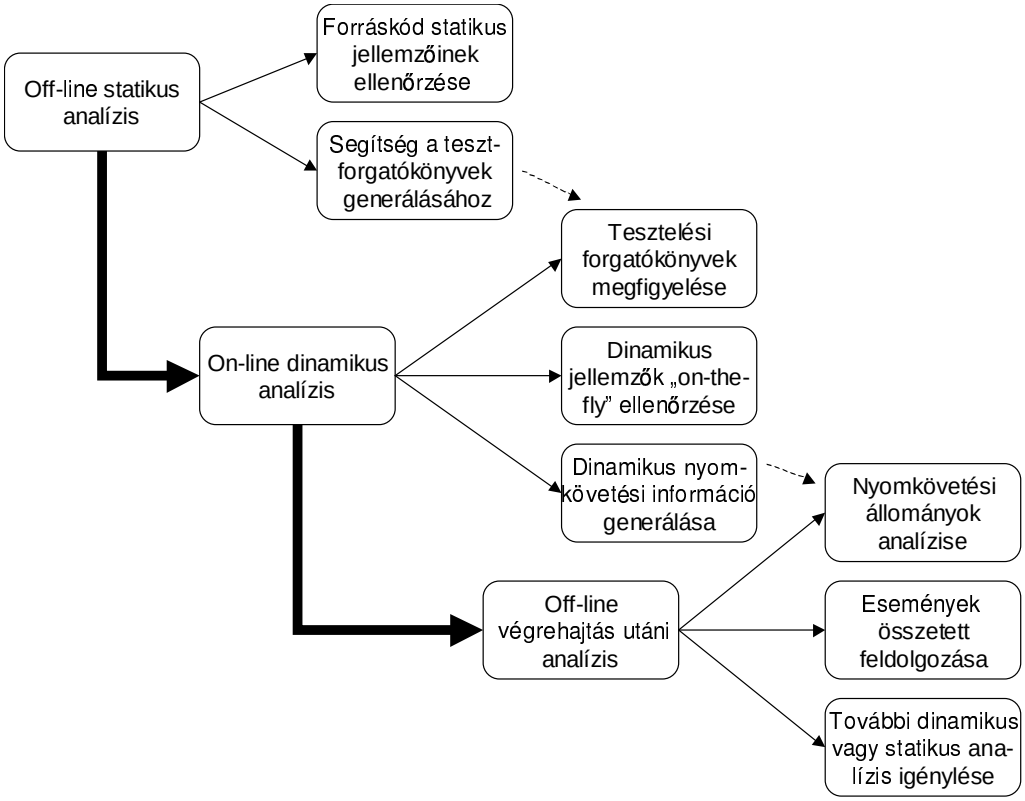
Elosztott hibakeresési megközelítések osztályozhatóak a fejlesztési ciklus fázisai szerint is (lásd 8. ábra).

5.2.1 Off-line elosztott hibakeresés statikus analízis alapján

A megközelítés a program kódját veszi alapul, így nem szükséges az aktuális program végrehajtása, és a program viselkedésének formális modelljére alapul, lehetővé téve bizonyos típusú tulajdonságok ellenőrzését, amiket általában temporális logikai formulákkal írunk le. A modell ellenőrzési technikák csak bizonyos tulajdonságok vizsgálatára használható, és nem adnak információt azokról a dinamikus tulajdonságokról, melyek a program aktuális futásától

függnek, mint például a terminálás. Továbbá hatalmas számítási költsége van általában a modellezett számítási tér összes megengedett állapotátmenetének megkeresésének.

Mindamellett az elosztott hibakeresés területén mégis hatalmas jelentősége van a programkód statikus analízisének, különösen, ha más kiegészítő megközelítésekkel kombináljuk össze.



8. ábra - Program analízis és hibakeresés a különböző fejlesztési fázisok során

5.2.2 On-line elosztott hibakeresés dinamikus analízis alapján

A statikus analízis már korábban említett korlátjai miatt, szükségünk van az on-line megközelítésre is, ami működés közben segít kiértékelni a program aktuális viselkedését. Ezek a megközelítések on-line megfigyelési technikán alapulnak, így törődniük kell a konzisztens globális állapotok akkurátus létrehozásának nehézségeivel. Amikor az adott program egy viselkedési mintáját már detektáltuk, ez a megközelítés megköveteli egy olyan vezérlési mechanizmus alkalmazását, ami segíti a felhasználót megvizsgálni az egyes (érdekes) számítási állapotokat. Továbbá ennek a megközelítésnek foglalkoznia kell a próba-effekttel is, hogy biztosítsa, a megfigyelt számítási útvonal ugyanazt a logikai viselkedést mutassa, mint az eredeti esetén, amikor még nem futott semmilyen megfigyelési mechanizmus (lásd 6. fejezet).

Dinamikus és statikus analízist kombinálhatjuk, hogy biztosítsuk azokat az elosztott hibakeresési funkcionalitásokat, amiket már illusztráltunk az „integrált tesztelés, aktív vezérlés és hibakeresés” megközelítés kapcsán.

5.2.3 Off-line elosztott hibakeresés végrehajtás utáni analízis alapján

A végrehajtás utáni analízis egy olyan megközelítés, ami hatékony módszert ad a program viselkedésének analízisére, mert a folyamatok lokális történeteinek előzőleg összegyűjtött nyomkövetési állományaira alapul. Egyrészt így könnyebbé válik a konzisztens globális állapotok létrehozása a kauzális precedencia láncok újragenerálásával. Ez csökkenti az on-line megközelítés által okozott futásidejű költségeket, valamint lehetővé teszi számos esemény-analizáló és vizualizációs eszköz segítségével a teljes számítási történet analízisét. Másrészt a végrehajtás utáni technikák integrálhatjuk on-line technikákkal is, hogy kihasználjuk a nyomkövetési, visszajátszási és hibakeresési módszereket, melyek a nem reprodukálhatóság problémáját oldják meg. Az on-line és végrehajtás utáni fázisokból álló inkrementális módszerek lehetőséget adnak a (gyakran előforduló) nagymennyiségű nyomkövetett információ kezelésére: az első futtatást csak a minimális mennyiségű információ begyűjtésére használjuk fel, ami biztosítja a reprodukálható újrafuttatást, majd később a futás utáni analízis segítségével meghatározhatjuk, vajon szükséges-e további információk begyűjtése a következő futások alapján.

A 8. ábrán összegezve látható, hogy a fejezetben tárgyalt megközelítések hogyan egészítik ki egymást.

Off-line: bizonyos jellemzők ellenőrzése statikus analízissel, és segít azonosítani a teszteléshez a releváns forgatókönyveket (*scenario*).

On-line: dinamikus jellemzők futás közbeni (*on-the-fly*) ellenőrzése, és az aktívan ellenőrzött futás közben a tesztelési forgatókönyvek megfigyelése.

Végrehajtás utáni: a nyomkövetett teljes globális történet analízise, összetettebb esemény-feldolgozás végrehajtása (pl. magas szintű esemény absztrakciók) és vizualizáció. Az analízis eredményeinek felhasználása, hogy meghatározzuk a további futtatásokat és a dinamikus analizálást.

5.3 Megfigyelési modellek

Végezetül az elosztott hibakeresési megközelítések csoportba sorolhatóak az elosztott számítás megfigyelési modellje alapján (lásd 9. ábra).

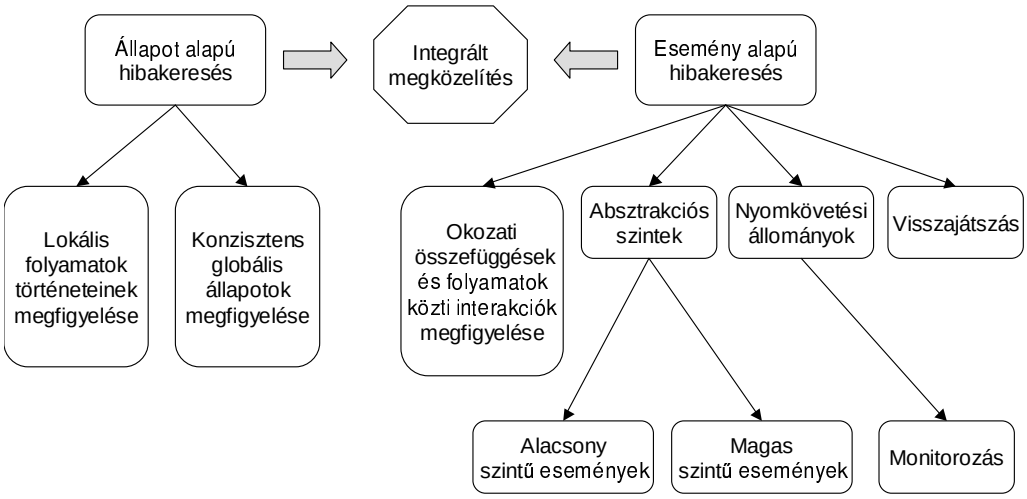
5.3.1 Állapot alapú nézetek az elosztott hibakereséshez

A megközelítés célja, hogy ugyanolyan funkcionalitásokat biztosítson, mint az állapot alapú szekvenciális hibakeresés, de a konzisztencia problémákat is próbálja megoldani. Két szinten foglalkozik az állapot feltérképezéssel:

A „komponens szinten[6]" a folyamatokat és a szálakat különálló számítási egységeknek tekintjük, melyek szekvenciális állapotátmeneteit kell megvizsgálnunk.

Az „elosztott alkalmazás szint" a komponensek közötti interakciókat globális szinten tekinti. Tipikusan egy elosztott memória-modellben a globális folyamat-interakciók az üzenetküldés vagy az RPC/RMI hívás. Egy számítási modellben, ami elosztott folyamatokat és többszálas végrehajtást is támogat, a folyamatokon belüli szálak közötti közös memóriás interakciók is számításba kell venni.

Formálisan egy állapot alapú nézet úgy értelmezi a folyamatok kommunikációját, mint az egyik folyamatnak a másik állapotára hatásának módja. Például néhány szerző úgy értelmezi az üzenetvételi műveletet, mint a küldő folyamat egy változó értékének (küldő pufferének) hozzárendelését a fogadó folyamat pufferének változójához. Így megtehetjük, hogy közvetlenül kifejezzük a függőségeket és a helyességi feltételeket az állapotváltozók segítségével.



9. ábra - Megfigyelési modell szerinti hibakeresési megközelítések

Az állapot alapú elosztott hibakeresés megpróbálja explicit kezelni a megfigyelési problémát az elosztott program előrehaladásának leírásával (a globális állapotok segítségével). Habár ez a megközelítés rendkívül fontos a hibák azonosításához, nehézzé teszi a logikai helyességi feltételek, az elosztott program kódjának helyének, és az aktuálisan megfigyelt számítási állapotok közti kapcsolatok meghatározását.

Az explicit esemény alapú megközelítés alkalmazhatóbb (mint az állapot alapú), mind a dinamikus elosztott program előrehaladásának, mind a helyességi tulajdonságok átlátszó értelmezésének biztosításához.

Ennek ellenére az állapot alapú nézetnek fontos szerepe van az elosztott hibakeresésben, hiszen megengedi a lokális és globális állapotok vizsgálatát.

[6] Nem azonos a komponens alapú szoftverfejlesztésben szereplő komponens fogalommal.

5.3.2 Esemény alapú nézetek az elosztott hibakeresésben

Az elosztott hibakeresésben számos szempontból nagy jelentőséggel bírnak az esemény alapú megközelítések.

Az *esemény* a *Lamport* által javasolt precedencia kapcsolatra alapozva egy természetes koncepció az ok-okozati kapcsolatokat nyomon követésére. Ezek a gyökerei számos munkának is: mind az elosztott számítások elméletének, mind a kauzalitási mechanizmusok (pl. vektor dátumbélyegek) támogatásának területén.

Az esemény alapú modellek lehetővé teszik az elosztott számítások különböző absztrakciós szinteken történő értelmezését. Megkönnyítik az elosztott programozási modell magas szintű absztrakciója és az elosztott rendszer alacsony szintű absztrakciója közötti megfeleltetést. Így lehetővé válik az esemény alapú modell használata, hogy a program elvárt viselkedését felhasználói szinten specifikáljuk, és ellenőrizhető legyen, hogy egyezik-e a megfigyelt program viselkedésével. Elosztott hibakereső eszközökre vonatkozóan az esemény absztrakciók biztosítják, hogy a felhasználó egy magas szintű nézetet kapjon az alkalmazás szintű modellel, pl. grafikus jelölésrendszert, vagy magas szintű szemantikát.

Az esemény egy alkalmas koncepció arra, kezeljük a reprodukálhatóság problémáját a determinisztikus visszajátszási technika által.

Az események áthidalják az elméleti koncepciók és a gyakorlati eszközök közötti szakadékot. Nevezetesen az esemény alapú modelleket könnyű kombinálni a nyomkövetés alapú monitorozással, az elosztott hibakereséssel és vizualizációs technikákkal.

Összegezve, az esemény alapú modellek olyan koncepciót adnak, hogy megértsük az elosztott programok szemantikáját, és megkönnyítik az elosztott hibakereső eszközök fejlesztését:

- Kapcsolódnak az elmélethez: kauzális precedenciához és az elosztott számítás részleges rendezéséhez.
- Különböző absztrakciós szinteken specifikálhatjuk a viselkedést.
- Nyomkövetést használva lehetővé teszik a reprodukálhatóságot.
- Megengedik a releváns (töréspontokhoz kapcsolódó) feltételek futásidejű detektálását.
- Kapcsolódnak az analízis és vizualizációs eszközökhöz, így áthidalva az elmélet és gyakorlat közti különbséget.
- Megengedik a monitorozó, nyomkövető megközelítések egységesítését.

5.4 Az esemény és állapot alapú elosztott hibakeresés integrálása

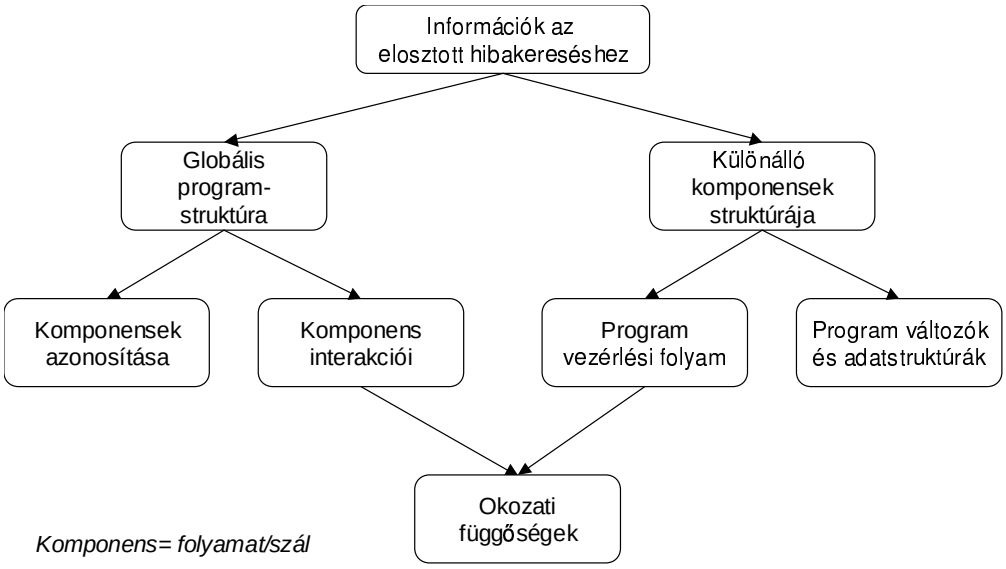
Az esemény alapú modell használható a program viselkedésének specifikálására, és a program bizonyos viselkedésének detektálására. Majd ezután szükséges a globális számítási állapotok vizsgálata, ezért válik természetessé az esemény és állapot alapú elosztott hibakeresési technikák összekapcsolása. Az

egyes folyamatok lokális történeteinek átvizsgálása állapot alapú hibakeresési technika segítségével történhet, ahogy azt a tipikus szekvenciális hibakeresők is felajánlják. Globális állapot nézeteknek csak a jelentősebb események detektálásakor kell létrejönnie, így az állapotvizsgálat rávilágíthat a hiba okára.

Az integráció legfőbb nehézség annak köszönhető, hogy szükség van az elosztott számítás konzisztens globális állapotainak (re)konstruálására valahányszor valamilyen jelentős eseményt detektáltunk, és csak ezután képes a felhasználó az egyes folyamatok történetét állapot alapú hibakeresési technikára alapozva megvizsgálni. A probléma megoldását a következő fejezetben tárgyaljuk.

6 Hibakeresési technikák

Az elosztott hibakeresés főbb technikáit céljaik szerint is osztályozhatjuk: a megfigyelés vagy az elosztott számítások vezérlése. A következő alfejezetekben áttekintjük a létező technikák legfontosabb jellemzőit.



10. ábra - Információk az elosztott hibakereséshez

6.1 Megfigyelés

Itt a megfigyelés koncepciójának egy szélesebb értelmezését tételezzük fel, lefedve a különböző területeket a számítás aktuális futásának alacsony szintű megfigyelésétől kezdve, az elosztott program viselkedésének logikai megfigyelésén át a globális számítási állapotok konzisztens megfigyeléséig. Az elosztott hibakeresésben az összes ilyen megfigyelési dimenziót figyelembe kell vennünk, hogy a program logikai jellemzőit tanulmányozni lehessen.

A megfigyelési technikák általános céljai:

(i) A szükséges információk összegyűjtése ([33], [55] 6. fejezete).

(ii) A létrehozott konzisztens globális állapotok kiértékelése a helyességi feltételekre.

(iii) Értelmezni, analizálni, vizualizálni és animálni az elosztott program viselkedését különböző absztrakciós szinteken ([55] 2. és 8. fejezete)

Általánosságban egy elosztott hibakereső eszköznek különböző típusú információkat kell, hogy szolgáltasson, ahogy azt a 10. ábra mutatja. A komponens és globális alkalmazás szintjének nézetét már elmagyaráztuk. A 10. ábra az elosztott hibakeresésre fókuszál, a szignifikáns információk megfigyelésére, hogy megértsük az okozati függőségeket, amik megmagyarázzák a program viselkedését.

Az információt a programfejlesztés számos fázisában megvizsgálhatjuk. A nyomkövetés alapú megközelítés által nyerjük ki a minimális információt a folyamatok interakcióról és a program vezérlési folyamáról, míg a globális számítási állapotokról és az egyes folyamatok állapotairól a fennmaradó információkat általában már dinamikusan nyerjük ki, interaktív hibakeresési ciklust alkalmazva[7]. Gyakorlati jelentősége miatt a következőkben röviden ismertetjük a nyomkövetés alapú technikákat az elosztott hibakeresés szemszögéből.

6.1.1 Nyomkövetési technika alkalmazása elosztott hibakeresésre

Az esemény alapú technikák monitorozáson alapulnak, hogy összegyűjthesse a releváns információkat, valamint nyomkövetési állományokat generálhasson, ami leírja az elosztott számítás történetét (lásd [55] 6. fejezet).

A nyomkövetés az alábbi fontosabb funkcionalitásokat biztosítja az elosztott hibakereséshez (lásd 11. ábra):

a) Lehetővé teszi a determinisztikus újrafuttatást [30][54].

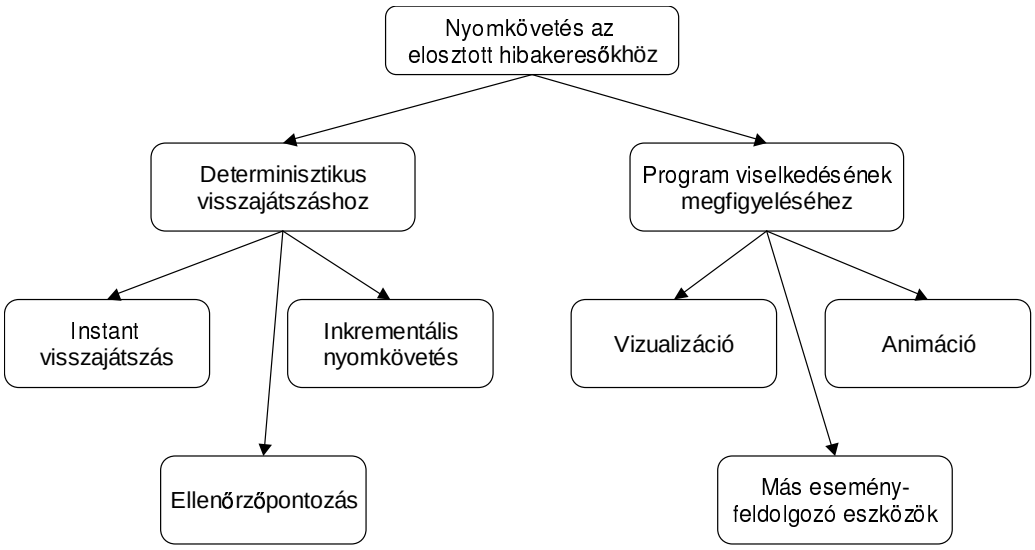
b) Elégséges információt gyűjt össze, hogy más eszközök számára lehetővé tegye a további eseményanalízist, segítve az elosztott program viselkedésének értelmezését, vizualizációját és animációját egy alkalmas absztrakciós szintjen.

Egy kritikus pont, hogy miképp határozzuk meg a szükséges információt, amit nyomon kell követni.

A determinisztikus újrafuttatásra vonatkozóan egy konzervatív megközelítés az *esemény-naplózás* (event log approach), amikor lementjük a folyamat teljes állapotinformációját, magában foglalva a változók értékét és a váltott üzenetek tartalmát is. Továbbá információkat tartalmaz még a folyamatokon

[7] Ez nem az az eset, amikor elosztott valós-idejű (real-time) alkalmazásokban keresünk hibát, mert az interaktív hibakeresés nem alkalmazható a valós-idejű kényszerek miatt. Valós-idejű elosztott hibakereséssel itt nem foglalkozunk (lásd [10], [53]).

belüli nem-determinizmus lehetséges forrásairól, mint például a valós óra értékéről, a külső I/O eszközökkel történő kommunikációról, és a belső nem-determinisztikus választásokról is. Ez még általánosabb esetekben is lehetővé teszi a determinisztikus újrafuttatását, és rendelkezik azzal az előnnyel, hogy elkülönítve megismételhetjük egy különálló folyamat (vagy folyamat-csoport) végrehajtását, biztosítva a nyomkövetett bemeneti információkat a külső környezetből. A megközelítés gyakran nagymennyiségű nyomkövetett információt hoz létre, és a legfőbb kockázat, hogy így növeli a memória és háttértárterület méretével kapcsolatban a követelményeket, és az aktív monitorozási tevékenység miatt továbbnöveli az alkalmazásba történő beavatkozás mértékét.



11. ábra - Nyomkövetés az elosztott hibakereséshez

Egy olyan alkalmazásban, ahol az egyes folyamatok determinisztikus viselkedést mutatnak, elégséges, ha csak az elosztott számítás releváns eseményeinek sorrendjét tároljuk el, a teljes állapotinformációk helyett. Ezt i*nstant replay* [30] technikának nevezik, ami az elosztott hibakeresés területén mérföldkőnek számít. Egyszerűen megfogalmazva, az *instant replay* megköveteli az összes folyamat újrafuttatását, de lehetővé teszi, hogy interaktív hibakeresési parancsokat használva tüzetesebben megvizsgáljuk minden egyes folyamat állapotát és a megfigyelt elosztott számítási útvonal eseményeit.

Hosszan futó alkalmazásokhoz ellenőrzőpont (*checkpoint*) technika alkalmazására lehet szükség, hogy lehetővé váljon a visszajátszás a számítás egy közbenső pontjától kezdve is (a kezdete helyett). Egy ilyen szolgáltatáshoz kell egy mechanizmus, ami konzisztens globális ellenőrzőpontokat készít (a megfelelő konzisztens globális állapothoz).

Az elosztott program magas szintű megfigyelését és viselkedésének analízisét támogató nyomkövetéshez sokkal több információt kell összegyűjteni, hogy megoldható legyen a program viselkedésének analíziséhez és vizualizációjához kapcsolódó utófeldolgozás. Amellett, ha van egy alkalmas eseménymodellezési absztrakciónk, a felhasználó meg tudja határozni az

absztrakció szintjét, amire az alkalmazás fejlesztésének minden egyes pontján szüksége lehet. Ez lehető teszi, hogy szűrjük, csoportba foglaljuk vagy elutasítsunk bizonyos típusú eseményeket és/vagy folyamatokat, így hozzájárulva a nyomkövetett információk mennyiségének csökkentéséhez. Ha a további analízis során feltárjuk, hogy további információkra van szükségünk, akkor lehetséges, hogy az elosztott programot determinisztikus módon újrafuttassuk és begyűjtsük ezeket a további szükséges információkat. Ezt az ún. inkrementális nyomkövetési szolgáltatást számos elosztott hibakereső eszköz biztosítani tudja [38]. Jó kompromisszum érhető el az *event logging* és az *instant replay* megközelítése között az inkrementális nyomkövetési technika alkalmazásával a felhasználói igények által irányított, determinisztikus visszajátszásra alapuló interaktív hibakeresési ciklus folyamán.

Az off-line vagy futás utáni feldolgozó eszközökhöz be kell gyűjteni az összes szükséges információt egy futás során, vagy a inkrementálisan nyomkövetett futások sorozata közben. On-line feldolgozás esetén (pl. vizualizáció) az eszközök a futás során generált eseményeket dolgozzák fel, ezért kevesebb szükség van nagy nyomkövetett eseményfájlok elmentésére.

Sajnos még az instant visszajátszási technika esetén sem lehet elkerülni a próba-effektet, tehát fennáll annak a lehetőségnek a kockázata, hogy a megfigyelt és szabadon futó program viselkedése között különbség legyen. Néhány szerző a szoftverek monitorozásához javasolta, hogy tartsák meg a monitorozáshoz szükséges kódrészleteket (*instruments*) az alkalmazásban mindvégig, így nem lesz ilyen különbség a futások között. Ez csak akkor járható út, ha a beavatkozás minimális, és megfelel az alkalmazás teljesítménykövetelményeinek egyaránt (lásd [55] 6. fejezete).

Az elmúlt évtizedekben intenzív kutatási téma volt a nyomkövetéshez szükséges idő és memóriaigény csökkentése [12][39][40][45]. Egy jellemező példa, hogy megpróbálják elkerülni azoknak a szükségtelen események nyomkövetését, melyek tudottan determinisztikus programtartományokhoz tartoznak.

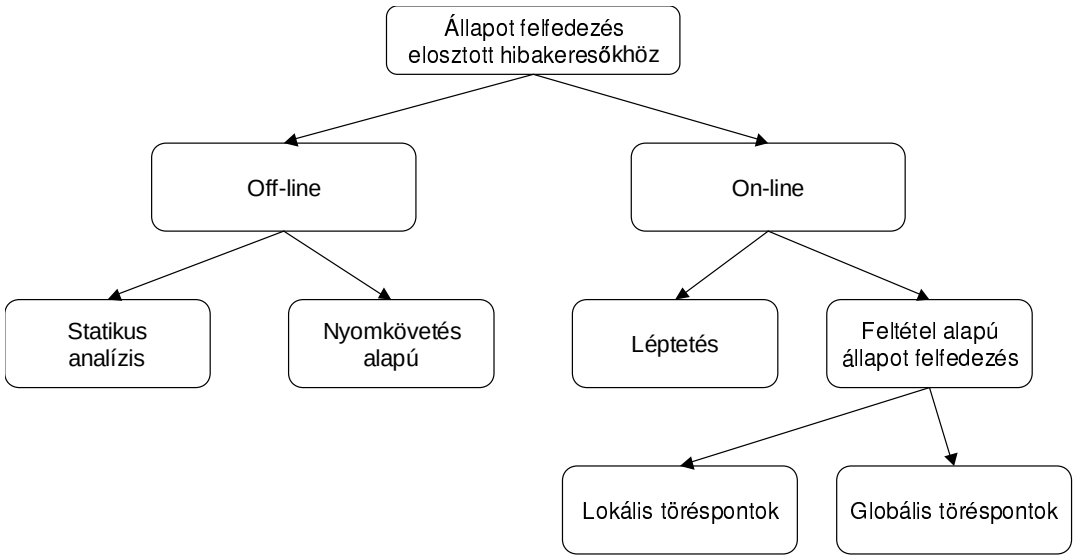
6.1.2 Állapot felfedezés az elosztott hibakeresésben

Egy elosztott hibakereső eszközzel szemben támasztott egyik legfontosabb követelmény az elosztott program számítási állapotainak felfedezése (lásd 12. ábra).

A Neumann gép megjelenésétől kezdve egy azonnali (*instant*) megfigyelési technikát alkalmaznak a klasszikus szekvenciális hibakeresésben: a memóriakép lementést (*memory dump*). Szekvenciális hibakeresés viszonylatában a technika legfőbb kritikája: az alacsony szintű absztrakciója és a módszer által generált információ mennyisége. Azonban a memória lementési technika az állapot alapú szekvenciális hibakeresés alapja is, de a lementett információt magasabb szinten mutatja be, és strukturálja az érdekes programentitások szerint (pl. verem, stb.). Az elosztott hibakeresés viszonylatában, a futó elosztott számításról egy ilyen típusú globális pillanatkép elkészítésére alkalmas technika kidolgozása nem egyszerű feladat, ahogy ezt már korábban taglaltuk. Vitatott, hogy a túl sok fellépő állapot miatt ez a technika limitáltan használható-e az elosztott hibakeresésben.

Egy léptetési szolgáltatás, ahogy azt a szekvenciális hibakeresésnél megtaláljuk, nem biztos, hogy hasznos az elosztott hibakeresésben is, habár elméletileg lehetséges elérni a konzisztens vágások egymásutánjával (lásd 2. fejezet). A probléma az, hogy az alternatív konzisztens futásoknak túl nagy az állapottere, amit ezzel a módszer alkalmazásával lépésről-lépésre fel kellene fedezni. Még ha ez megközelítés használatos magas szintű absztrakciókhoz (grafikus programozási nyelvekhez) kapcsolva, ami az interakciós eseményekre próbál fókuszálni, még mindig nem alkalmazható valós méretű alkalmazásokra.

A nyomkövetési állományok értelmezése alapján az off-line vizualizációs eszközök hozzájárulhatnak, hogy felismerjük a program viselkedésének érdekes mintáit. Interaktív, a felhasználó által vezérelt vizualizációs eszközök lehetővé tehetik a felhasználó számára, hogy válasszon az absztrakciós szintek között és a program fontosabb régiói között, melyet az eszköznek figyelembe kell venni.



12. ábra - Állapot felfedezés az elosztott hibakeresésben

A feltétel alapú állapot felfedezés egy olyan technika, ami lehetővé teszi a felhasználó számára, hogy olyan logikai feltételeket adjon meg, melyek ha egy bizonyos állapotban igazzá válnak, akkor azon a ponton a program végrehajtása felfüggesztésre kerül. A cél az, hogy a felhasználó szelektív módon tudja az állapotok felfedezését elvégezni, hogy elkerülje a számítás állapotterének kimerítő bejárását. Ez a megközelítés fontos, mivel a teljesen automatikus elosztott hibakeresés gyakorlatilag kivitelezhetetlen, és a felhasználó így szabad kezet kap állapottér bejárásának irányítására.

A szekvenciális programokban ezek a feltételek általában programváltozókat, állapotokat, vagy az instrukciók elhelyezkedését foglalják magukban. Az így létrejövő töréspontoknak egyszerű, jól definiált szemantikájuk van a szekvenciális programokban, de az elosztott programokban számos lehetséges értelmezésére nyílik lehetőség. Mindamellett a töréspontok és az interaktív hibakeresési technikák rendkívül intenzíven és megjósolhatatlan következményekkel szólnak bele a program viselkedésébe. Így az elosztott

hibakeresésben a töréspontok kombinálása determinisztikus visszajátszási technikákkal rendkívül fontossá váltak.

A globális feltételeket számos formában fejezhetjük ki. Kifejezhetjük a változók, kódlokáció vagy az elosztott program egyes folyamatainak lokális állapotainak szempontjából. Egy lokális kijelentés egy adott folyamatra vonatkozik, tehát megfelel egy lokális töréspontnak, ami könnyen detektálható egy elosztott hibakereső eszköz által. Egy globális kijelentés egy *boolean* kifejezés különböző feltételeket tartalmazva, általános formáiban a lokális kijelentések „ÉS" kapcsolata (konjunktív globális kijelentés) vagy lokális kijelentések „VAGY" kapcsolata (diszjunktív globális kijelentés). Léteznek más típusú globális kijelentések, melyek az absztrakt események temporális kapcsolatát fejezik ki.

A legtöbb létező elosztott hibakereső csak lokális töréspontok megadását teszi lehetővé. Ez annak köszönhető, hogy nehézségeket okoz az általános alakú globális kijelentések kiértékelése, ahogy azt már megtárgyaltuk. Valójában hatékony detektáló algoritmusok csak rendkívül kötött formájú globális kijelentésekre léteznek.

Emellett még az egyszerű lokális töréspontok esetén is szükséges, hogy eldöntsük mi is történjen, amikor egy adott folyamat futtatása során egy lokális feltételt detektáltunk. On-line megközelítésben a kérdés úgy vetődik fel, hogyan állítsuk meg az elosztott alkalmazást egy konzisztens globális állapotban, ami értelmes a detektált feltétel kiértékeléséhez. Tipikusan két fő megközelítést létezik [21][37]:

(i) Stop üzenet küldése az összes többi folyamatnak.

(ii) Várni mindaddig, amíg az összes többi folyamat „természetesen" blokkolódik: bemeneti információra várva vagy a kezdeményező folyamattal szinkronizációra.

Mindkét esetben késleltetés lép fel a többi folyamatban a leállítási feltétel detektálásáig. Ez egy előre meg nem jósolható késleltetés, így ez a megközelítés csak akkor alkalmazható, ha garantálható, hogy más folyamat nem lehet hatással a detektált feltétel vizsgálatára. Ez természetesen függ a feltétel típusától, de a fenti megközelítés lokális kijelentésekhez alkalmazhatónak tűnik.

Globális töréspontoknál a feltételek globális kijelentésekhez kapcsolódnak, így olyan detektáló algoritmus szükséges, ami rákényszeríti a feltételben szereplő folyamatokat, hogy megálljanak, míg a többi folyamat folytathatja futását, mint az előző esetben. Vegyünk például egy 4 folyamatból (P_1, P_2, P_3, P_4) álló elosztott programot, és a globális töréspont tartalmazzon változókat P_1 és P_2 folyamatból, de P_3 és P_4 folyamatból nem. A P_1 és P_2 folyamatokat rákényszerítjük, hogy megálljanak a globális törésponton, de P_3 és P_4 folyamatok folytathatják futásukat. Általánosságban, P_3 és P_4 folyamatok hatással lehetnek a globális rendszerállapotra, (pl. üzenetküldéssel vagy fogadással), íly módon sajnos megváltoztathatják azokat a feltételeket, amik megmagyaráznák az esetleges hibás szituációkat.

Egy értelmes globális állapot rekonstrukciójához szükség lehet visszapergetési és ellenőrzőpont technikákra. Ez komoly problémákat szülhet, ha nem visszajátszás alapú megközelítést követtünk.

A visszajátszás alapú megközelítés esetén töréspontokat csak a visszajátszás során helyezünk el. Hogy kezeljük ezeket a szituációkat, számos visszajátszás alapú hibakereső eszközöket javasoltak [32], amik képesek rekonstruálni egy konzisztens globális állapotot, azaz minden folyamat az utolsó lokális állapotán áll, és ahol a bevont folyamatok (mint például P_1 és P_2) a globális kijelentést kielégítik. A globális állapot rekonstrukciója itt a nyomkövetésen alapul, hogy felismerjük a szükséges leállást kiváltó lokális állapotokat. Néhány javaslatban új jelölésekkel ellátott nyomkövetési állomány generálódik, hogy az elosztott programot az utolsó konzisztens globális állapotig visszajátszhassuk, amit az újonnan bevezetett direktívák határoznak meg.

A globális töréspontho z tartozó utolsó konzisztens globális rekonstrukciója mellett még szükséges lehet az egyes folyamatok lokális állapotainak vizsgálatára is. Ezt elérhetjük, ha integrálunk egy esemény alapú elosztott hibakereső eszközt (ami támogatja a globális töréspontokat és a konzisztens globális állapotokon történő megállást) és egy állapot alapú szekvenciális hibakereső eszközt, amit elkülönülten alkalmazhatunk az alkalmazás bármely folyamatára.

6.2 Vezérlés

Az elosztott számítást vezérelnünk kell, hogy az elosztott hibakeresés során az alábbi célokat elérhessük (lásd 13. ábra):

(i) A nem-determinizmus kezelése.

(ii) Adott elosztott számítási útvonalra kényszerítés.

(iii) Adott globális kijelentések kényszerítése.

Az elosztott programok nem-determinisztikus viselkedése miatt egyszerűen a program „szabad” megismétlésével nem tudjuk a többszörös végrehajtás igénylő tesztelést kivitelezni. Ez determinisztikus újrafuttatást igényel, aminek a technikája a passzív vezérlés egy formájára alapul, mivel az újrafuttatás csak vakon utánozza az előző futtatás viselkedését.

A vezérlés egy aktív formájára is szükség van (*kontrollált végrehajtás* vagy *aktív kontroll* [50]), mivel alternatív elosztott számítási útvonalakat is be kell járnunk, hogy megvizsgálhassuk a program dinamikus viselkedését. A felügyelő folyamat vezérli az újrafuttatáshoz az események sorrendjét, pl. különböző üzenetkézbesítési sorrenddel, vagy alternatív folyamat ütemezések által, hogy megvizsgálhassuk a nem-determinisztikus végrehajtás következményeit.

A kívánt eseménysorrendeket esemény-kifejezésekkel vagy parancs szkriptekkel adhatjuk meg, melyeket az elosztott hibakereső értelmez. A cél az események összes/valamennyi lehetséges permutációjának generálása és tesztelése. Az aktív vezérlés ilyenformán lehetővé teszi, hogy az elosztott hibakeresés kiegészítő szerepet játsszon a statikus analizáló és tesztelési eszközök mellett, megkönnyítve a helyességi tulajdonságok kiértékelését az elosztott számítási útvonalak terének felfedezésével. Az ilyen útvonalak specifikálása történhet:

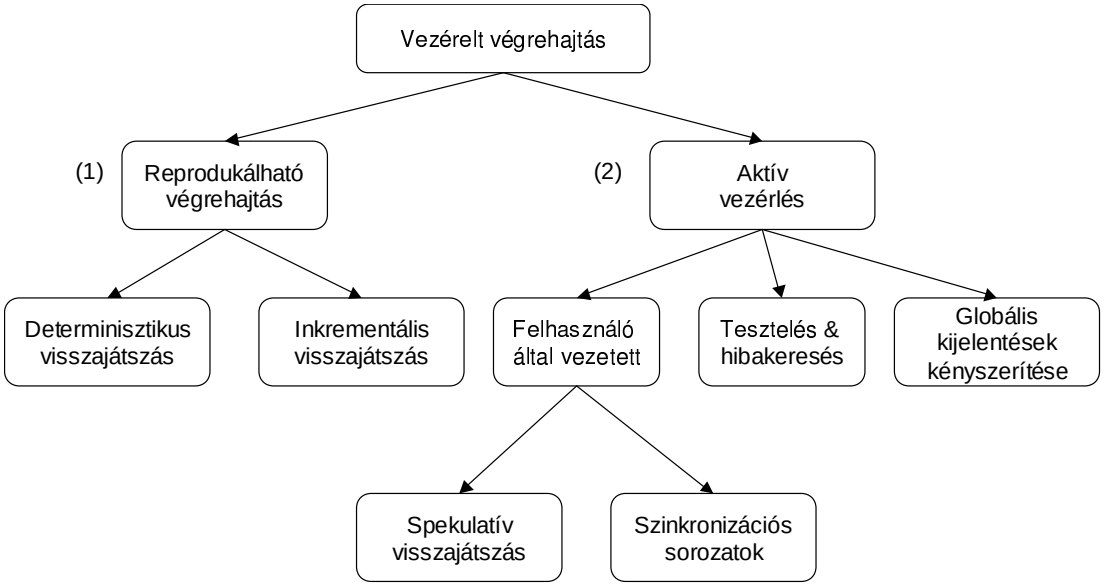
(i) Felhasználó által.

(ii) Automatikus tesztelési eszközzel generálva.

(iii) Automatikusan kényszerítve egy globális kijelentés kiértékelő/vezérlő által.

Egy példa az (i) megközelítésre a spekulatív visszajátszási technika [48]. A spekulatív visszajátszás során a felhasználó bármelyik lehetséges konzisztens globális állapotot kiválaszthatja és megkérheti az elosztott hibakereső eszközt, hogy a programot hajtsa végre újra egy üzenet naplófájlt használva. Így a felhasználó detektálhatja a különböző eseménysorrendek milyen különböző megfigyelhető viselkedést hozhatnak létre. Majd a felhasználó további folyamatok közi függőséget vezethet be, és újrafuttathatja a programot az új szinkronizációs kényszerek mellett.

Egy másik példa a *szinkronizációs szekvenciák* használata a *determinisztikus végrehajtás tesztelés* [49] módszerben. Ez egy adott bemenet mellett a konkurens program végrehajtását kényszeríti, hogy a felhasználó által definiált szinkronizációs parancsok sorozata szerint hajtódjon végre. A technikának egy másik használata már az előző alfejezetben említésre került, amikor az elosztott hibakereső eszköz megjegyzésekkel ellátott nyomkövetési állományt generált.



(1) Követi / utánozza az előzőleg végrehajtott útvonalat
(2) Új generált útvonalra kényszerít

13. ábra - Vezérelt végrehajtás

A második megközelítésre egy példát már tárgyaltunk az előző fejezetben a „tesztelés, aktív kontroll és hibakeresés” módszeréhez kapcsolódóan. Ez a megközelítés az alapja a SEPP projektekben [31] végzett munkának a tesztelési és hibakereső eszközök integrálása kapcsán, amit az [55] 9., 13. és 16. fejezete mutat be.

A harmadik megközelítés kapcsolódik a globális kijelentések automatikus kiértékeléséhez, amit már szintén tárgyaltunk az elosztott hibakeresési módszerek

fejezetben. Ennek egy elosztott hibakeresési ciklusba történő integrálása csak azt követeli meg, hogy a felhasználó globális kijelentésekkel megadja a program helyes viselkedését. Ezek után már az elosztott hibakeresőtől függ, hogy monitorozza a végrehajtást, és detektálja vagy kényszerítse ezeknek a logikai kondíciók kielégítését. A felhasználó ekkor megvizsgálhatja a konzisztens globális állapotot, több információt kapva a hiba okáról. A hibakeresési ciklus ezen pontján a felhasználó a program viselkedésével kapcsolatban olyan új információk birtokába juthat, amik lehetővé teszik, hogy újabb globális tulajdonságokat jelentsen ki, amit a globális kijelentés vezérlő által kell rákényszeríteni a programra [50][52]. Ezután egy aktív vezérlési fázis kezdődik, ahol a vezérlő felelős a feltételek kényszerítéséért, lehetővé téve más, potenciálisan hibás szituációnak a vizsgálatát. A harmadik megközelítés a legambiciózusabb: koncepcionálisan lehetővé tenné egy elosztott hibakereső vezérlőjének az igényelt szinkronizációs kényszerek automatikus érvényesítését. Sajnos csak néhány elosztott hibakereső eszköz képes adott globális kijelentések kényszerítésére, és csak nagyon kötött kifejezések elejéig [52].

7 Elosztott hibakereső rendszerek architektúrája

Az elosztott hibakereső rendszerek tervezésének meg kell felelnie az alapvető megfigyelési és vezérlési funkcionalitások által támasztott követelményeknek, amiket az előző fejezetekben már tárgyaltuk. A legtöbb elérhető párhuzamos és elosztott hibakereső eszköz architektúrája követi a p2d2 architektúráját [22], ami kliens-szerver modellre alapul. Mindamellett az elérhető hibakereső eszközök nagyon kifinomult vizuális és grafikus felhasználói felületeket nyújtanak, és széles körben lefedik az egyes elosztott folyamatok és szálak interaktív hibakereséséhez szükséges funkcionalitásokat.

A közelmúltban néhány elosztott hibakereső eszköz teljes párhuzamos szoftverfejlesztő környezetbe integrálva került kifejlesztésre. Mindazonáltal még mindig számos nyitott kérdést találunk a sikeres eszköz-integrációhoz kapcsolódóan [44].

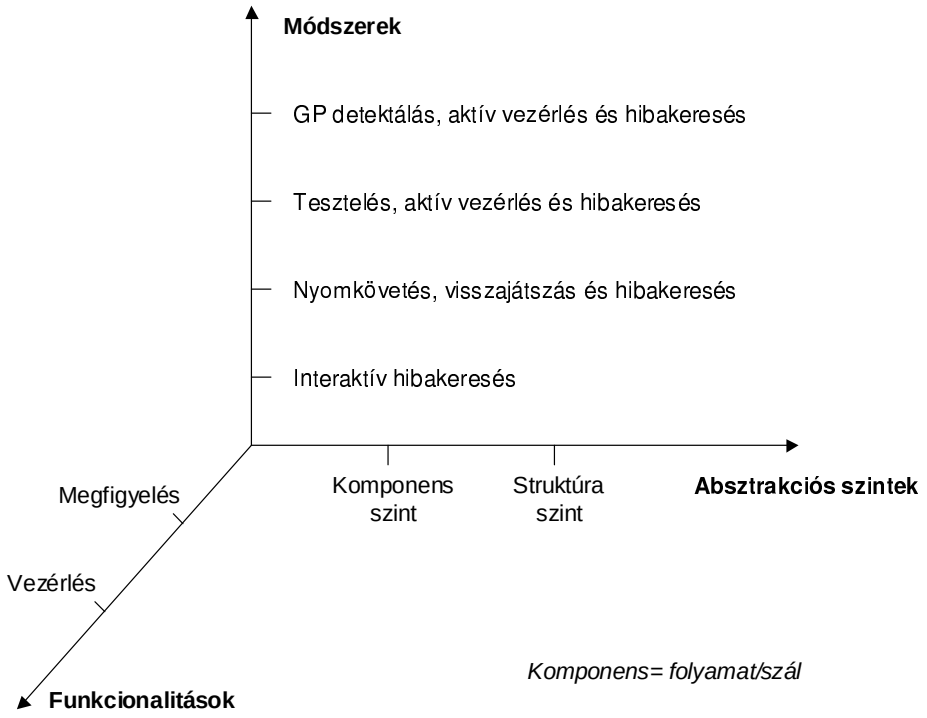
Ahelyett, hogy az architektúra a funkcionalitások egy fix halmazát támogatná, az elosztott hibakereső eszköz architektúrájának kiterjeszthetőnek és alkalmasnak kell lennie, hogy más párhuzamos szoftverfejlesztési eszközzel integrálhassuk össze. Szükséges, hogy több elosztott hibakeresési megközelítés és módszert lehessen támogatni egy közös architektúrájú keretrendszerben. A SEPP projekt ezt a megközelítést követte a DDBG hibakereső ([55] 13. fejezet) tervezésénél.

8 Összegzés

Az elosztott hibakeresési tevékenység még mindig hatalmas akadályokkal néz szembe, hogy befolyását növelhesse a felhasználókra. Ahogy a [20] egy *PTOOLS* riportra hivatkozva említi, a párhuzamos és elosztott alkalmazások fejlesztői (*PVM*-et használva) még mindig a klasszikus „printf” stílusú hibakeresést alkalmazzák. Továbbá senki sem tudja, hogy a maradék 10% valójában elégedett-e

a jelenleg elérhető kereskedelmi vagy akadémiai elosztott hibakereső eszközökkel. Ebben a segédletben az elosztott hibakeresés legfontosabb dimenzióit tárgyaltuk meg. Az elosztott hibakeresés alapjait kapcsolatba hoztuk az elosztott számítás elméletével, hogy megmutassuk az elosztott hibakeresés nehézségeit, és megmagyarázzuk miért nem működik az elosztott hibakeresés során a naiv „printf" alapú megközelítés.

Számos módszer mutattunk, melyek illusztrálták a statikus analízisnek, a tesztelésnek, és a dinamikus analízisnek egymást kiegészítő szerepköreit az elosztott program viselkedésének megismerésének céljából. A 14. ábra összegzi az elosztott hibakeresés dimenzióit. A legfontosabb megfigyelési és vezérlési technikákat ismertettük, kiemelve az interaktív visszajátszás alapú elosztott hibakeresést, valamint azt, hogyan segíti elő egy hibakereső ciklus létrehozását, amiben a felhasználó ismételten megvizsgálhatja és inkrementálisan gyűjtheti az információt, így elősegítve az elosztott program viselkedésének analízisét. Egy ideális megközelítést szintén körvonalaztunk: a program helyességi tulajdonságainak specifikálásával kezdve, majd kísérletet téve azok illesztésére a megfigyelt viselkedésre. Az ilyen megközelítés az interaktív ciklikus visszajátszás alapú elosztott hibakeresési megközelítéshez kapcsolódik, hogy lehetővé váljon a felhasználó számára, hogy megvizsgálja a releváns globális számítási állapotokat.



14. ábra - Felhasználói nézetek

Az elmúlt évtizedben az elosztott hibakeresés területén történt kutatási és fejlesztési erőfeszítések rámutattak, hogy az elosztott hibakeresési tevékenységnek csak akkor van értelme, ha megfelelően integráljuk más kiegészítő eszközökkel az analízishez, a vizualizációhoz, a program megfigyeléshez és vezérléshez. Igen kifinomult önálló és autonóm elosztott hibakereső eszközök [11] mellett az

elosztott hibakereső eszközök következő generációja egyre inkább igényli a flexibilisebb és kiterjeszthetőbb elosztott hibakeresési architektúrát, hogy integrálni lehessen az elosztott hibakeresési funkcionalitásokat más eszközökkel. Ez a követelmény még fontosabbá válik a jelenlegi homogén és heterogén, párhuzamos és elosztott platformokon folyó fejlesztéseknél.

Remélhetőleg az elkövetkező években tanui lehetünk az elosztott hibakereső eszközök szabványos interfészeinek megjelenésének (HPDF kezdeményezés [14]). Tapasztalni fogjuk az elosztott hibakereső eszközök tervezésének és architektúrájának fejlődését, melyeket az újonnan megjelenő párhuzamos és elosztott programozási modellekre alkalmazhatunk, mint például a komponens alapú, vagy mobil számítási absztrakció.

Még ha ilyen nagy elvárásokkal kell is szembenézni a közeljövőben, az hibakeresési megoldások befolyása leginkább attól függ, hogy milyen sikeresen integrálják össze őket egy könnyen tanulható és használható eszközzé, olyan felhasználó-barát felülettel, mely találkozik a felhasználók gyakorlati igényeivel. Ezeket a témákat egyre gyakrabban tárgyalják a felhasználók, eszközfejlesztők és szoftvertervezők. A trend, hogy az emberi tényezőt, minél inkább figyelembe veszik a felhasználói felület és a hibakereső eszköz által nyújtott funkcionalitások tervezésekor.

9 Köszönetnyilvánítás

Az oktatási segédlet létrejöttéhez hozzájárult a Magyar-Portugál Kormányközi Bilaterális TéT Projekt, és nagymértékben épült a SEPP konzorcium eredményire, melyeket az [55] kiadványban tette elérhetővé az érdeklődők számára.

Külön köszönetet mondanék konzulensemnek Kacsuk Péternek, és portugál partnereiknek J. C. Cunha-nak és J. Lourenco-nak értékes észrevételeikért és támogatásukért.

10 Terminológia

Causal precedence	Kauzális precedencia
Checkpoint	Ellenőrzőpont
Consistent global cut	Konzisztens globális vágás
Consistent global state	Konzisztens globális állapot
Debugger	Hibakereső eszköz
(Distributed) debugging	(Elosztott) hibakeresés
Distributed computations	Elosztott számítások
Erroneous situation	Hibás helyezet
Event logging	Esemény-naplózás
Frontier of cut	Vágás front
Global/local predicate	Globális/lokális kijelentés
Local/global history	Globális/lokális történet
Non-deterministic behaviour/execution	Nem-determinisztikus viselkedés/ végrehajtás
Post-mortem	Végrehajtás utáni
Probe-effect	Próba-effekt
Process	Folyamat
Programming bug	Programozási hiba
Replay technique	Visszajátszási technika
Specification bug	Specifikációs hiba
Trace file	Nyomkövetési fájl
Tracing	Nyomkövetés

11 Referenciák

[1] Proceedings of the ACM Workshop on Parallel and Distributed Debugging, volume 24 of ACMSIGPLAN Notices. ACM, 1988.

[2] Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, volume 26 of ACM SIGPLAN Notices. ACM, 1991.

[3] Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, volume 28 of ACM SIGPLAN Notices. ACM, 1993.

[4] O. Babaoglu, E. Fromentin, and M. Raynal. A unified framework for the specification and run-time detection of dynamic properties in distributed computations. Technical Report UBLCS95-3, University of Bologna, June 1995.

[5] O. Babaoglu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In S. J. Mullender, editor, Distributed Systems, chapter 4, pages 55-96. Addison-Wesley, second edition, 1993.

[6] P. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. In Proceedings of Workshop on Parallel and Distributed Debugging, pages 11-22. ACM Press, 1988.

[7] M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. ACM Trans. Comput. Syst., 3(1):63-75, 1985.

[8] C. M. Chase and V. K. Garg. Detection of global predicates: Techniques and their limitations. Distributed Computing, 11(4):191-201, 1998.

[9] R. Cooper and K. Marzullo. Consistent detection of global predicates. In Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, volume 26 of ACM SIGPLAN Notices, pages 167-174. ACM Press, December 1991.

[10] P. S. Dodd and C. V. Ravishankar. Monitoring and debugging distributed real-time programs. Software--Practice and Experience, 22(10), 1992.

[11] Etnus Inc., Framingham, MA. TotalView User's Guide (v3.9.0), June 1999.

[12] A. Fagot and J. Chassin-de Kergommeaux. Optimized execution replay mechanism for rpc-based parallel programming models. Technical report, LMC-IMAG, Grenoble, France, 1995.

[13] C. J. Fidge. Partial orders for parallel debugging. In Proceedings of Workshop on Parallel and Distributed Debugging, volume 24 of ACM SIGPLAN Notices, pages 183-194. ACM, 1988.

[14] J. Francioni and C. M. Pancake. High performance debugging standards effort. URL: http://www.ptools.org/hpdf/.

[15] J. M. Francioni. Debugging and Performance Tuning for Parallel Computer Systems, chapter Determining the Effectiveness of Interfaces for Debugging and Performance Analysis Tools, pages 127-142. IEEE Computer Society Press, 1996.

[16] P. Fritzson, editor. Automated and Algorithmic Debugging, volume 749 of LNCS. Springer-Verlag, May 1993.

[17] E. Fromentin, M. Raynal, V. K. Garg, and A. Tomlinson. On the fly testing of regular patterns in distributed computations. In K. C. Tai, editor, Proceedings of the 23rd International Conference on Parallel Processing. Volume 2: Software, pages 73-76, Boca Raton, FL, USA, August1994. CRC Press.

[18] V. Garg and C. Chase. Distributed algorithms for detecting conjunctive predicates. In Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS'95), pages 423-430, Los Alamitos, CA, USA, May 30-June 2 1995. IEEE Computer Society Press.

[19] V. Garg, C. Chase, J. R. Mitchell, and R. Kilgore. Tools and Environments for Parallel and Distributed Systems, chapter Efficient Detection of Unstable Global Conditions Based on Mono-tonic Channel Predicates, pages 195-226. Kluwer Academic Publishers, 1996.

[20] G. A. Geist. Debugging and Performance Tuning for Parallel Computer Systems, chapter Visualization, Debugging, and Performance in PVM, pages 65-77. IEEE Computer Society Press, 1996.

[21] D. Haban and W. Weigel. Global events and global breakpoints in distributed systems. In: Bruce D. Schriver, editor, Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences, volume II (Software Track), pages 166-175. IEEE Computer So-ciety Press, January 1988.

[22] R. Hood and D. Cheng. Tools and Environments for Parallel and Distributed Systems, chapter Accomodating Heterogeneity in a Debugger - a Client- Server Approach, pages 175-194. Kluwer Academic Publishers, 1996.

[23] R. Hood, K. Kennedy, and J. Mellor-Crummey. Parallel program debugging with on-the-fly anomaly detection. In Proceedings of IEEE/ACM Supercomputing'90, pages 74-82. IEEE Computer Science Press, 1990.

[24] M. Hurfin, M. Mizuno, M. Raynal, and M. Singhal. Efficient distributed detection of conjunctions of local predicates. Technical Report TR-967, IRISA, 1995.

[25] M. Hurfin, N. Plouzeau, and M. Raynal. Detecting atomic sequences of predicates in distributed computations. ACM SIGPLAN Notices, 28(12):32-42, December 1993.

[26] P. Kacsuk. Macrostep-by-macrostep debugging of message passing parallel programs. In Proceedings of Tenth IASTED International Conference on Parallel and Distributed Computing and Systems, pages 527-532, Las Vegas, Nevada, USA, October 1998.

[27] P. Kacsuk. Systematic debugging of parallel programs based on collective breakpoints. In Proc. of International Symposium on Software Engineering for Parallel and Distributed Systems, pages 83-96, Los Angeles, California, May 1999.

[28] P. Kacsuk, R. Lovas, and J. Kovács. Systematic debugging of Parallel programs based on collective breakpoints and macrosteps. In Proc. of 5th International Euro-Par Conference, pages 90-97, Toulouse, France, August/September 1999. Springer-Verlag.

[29] L. Lamport. Time, clocks, and the ordering of events in a distributed system. CACM,21(7):558-565, 1978.

[30] T.J. LeBlanc and J.M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. IEEE Transactions on Computers, C-36(4):471-481, April 1987.

[31] J. Lourenco, J.C. Cunha, H. Krawczyk, P. Kuzora, M. Neyman, and B. Wiszniewski. An integrated testing and debugging environment for parallel and distributed programs. In Proc. 23rd EUROMICRO Conference, pages 291-298, Budapest, Hungary, 1997. IEEE ComputerSociety.

[32] Y. Manabe and M. Imase. Global conditions in debugging distributed programs. J. Parallel and Distributed Computing, 15:62-69, May 1992.

[33] D. C. Marinescu, J. E. Lumpp Jr., T. L. Casavant, and H. J. Siege. Models for monitoring and debugging tools for parallel and distributed software. J. Parallel Distributed Computing, 9,June 1990.

[34] F. Mattern. Virtual time and global states in distributed systems. In M. Cosnard, P. Quinton, M. Raynal, and Y. Robert, editors, Proc. Int. workshop on Parallel and Distributed Algorithms. North Holland, 1989.

[35] C. E. McDowell. Debugging and Performance Tuning for Parallel Computing Systems, chapter Race Detection - Ten Years Later, pages 101-126. IEEE Computer Society Press, 1996.

[36] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. ACM Computing Surveys, 21(4):593-622, 1989.

[37] B. P. Miller and J.-D. Choi. Breakpoints and halting in distributed systems. In IEEE, editor, Proceedings of International Conference on Distributed Computing Systems, 1988.

[38] B. P. Miller and J.-D. Choi. A mechanism for efficient debugging of parallel programs. In: Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation, volume 23 of ACM SIGPLAN Notices, pages 135-144. ACM Press, July 1988.

[39] R. H. B. Netzer. Optimal trace and replay for debugging shared-memory parallel programs. In: Proceedings of the 3rd ACM/ONR Workshop on Parallel and Distributed Debugging. ACM Press, 1993.

[40] R. H. B. Netzer. Trace size vs. parallelism in trace-and-replay debugging of shared-memory programs. LNCS, 768:617-??, 1994.

[41] R. H. B. Netzer, T. W. Brennan, and S. K. Damodaran-Kamal. Debugging race conditions in message-passing programs. In Proceedings of the Symposiun on Parallel and Distributed Tools SPDT'96, pages 31-40, 1996.

[42] C. M. Pancake. Debugging and Performance Tuning for Parallel Computer Systems, chapter Collaborative Efforts to Develop User-Oriented Parallel Tools. IEEE Computer Society Press, 1996.

[43] C. M. Pancake and S. Utter. A Bibliography of Parallel Debuggers. ACM SIGPLAN Notices, 26(1):21-37, January 1991.

[44] D. A. Reed, J. S. Brown, A. H. Hayes, and M. L. Simmons. Debugging and Performance Tuning for Parallel Computer Systems, chapter Performance and Debugging Tools: A Research and Development Checkpoint. IEEE Computer Society Press, 1996.

[45] M. A. Ronsse and D. A. Kranzlmuller. Rolt-mp: Replay of lamport timestamps for message passing systems. In Proceedings of Euromicro Workshop on Parallel and Distributed Processing, pages 87-93, 1998.

[46] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. Distributed Computing, 7(3):149-174, 1994.

[47] M. L. Simmons, A. H. Hayes, D. A. Reed, and J. Brown, editors. Debugging and Performance Tuning for Parallel Computing Systems. IEEE Computer Science Press, 1996.

[48] J. M. Stone. A graphical representation of concurrent processes. ACM SIGPLAN Notices, 24(1):226-235, January 1989.

[49] K.-C. Tai, R. H. Carver, and E. E. Obaid. Debugging concurrent ada programs by deterministic execution. IEEE Trans. Software Eng., 17(1):45-62, January 1991.

[50] A. Tarafdar and V. K. Garg. Predicate control for active debugging of distributed programs. In Proceedings of the 1st Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS/SPDP-98), pages 763-769, Los Alamitos, March 30-April 3 1998. IEEE Computer Society.

[51] A. I. Tomlinson and V. K. Garg. Detecting relational global predicates in distributed systems. In B. P. Miller and C. McDowell, editors, Proceedings of the 3rd ACM/ONR Workshop on Parallel and Distributed Debugging, volume 28 of ACM SIGPLAN Notices, pages 21-31, New York, NY, USA, May 1993. ACM Press.

[52] A. I. Tomlinson and V. K. Garg. Maintaining global assertions on distributed systems. In: International Conference on Computer Systems and Education, 1994.

[53] J. J. P. Tsai and S. J. H. Yang, editors. Monitoring and Debugging of Distributed Real-Time Systems. IEEE Computer Society Press, 1995.

[54] L. Wittie. Debugging distributed C programs by real time replay. In Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel

and Distributed Debugging, volume 24 of SIG-PLAN Notices, pages 57-67. ACM Press, January 1989.

[55] Parallel Program Development for Cluster Computing: Methodology, Tools and Integrated Environments, Editors: P. Kacsuk, J.C. Cunha and S.C. Winter, Nova Science, New York, 2001