Szoftverrendszerek tesztelési modellje

DR. SZIRAY JÓZSEF

Széchenyi István Egyetem
sziray@sze.hu

Reviewed

Kulcsszavak: szoftvertesztelés, verifikáció, validáció, hibamodellek, formális módszerek

A cikk komplex szoftverrendszerek tesztelésének általános szempontjaival foglalkozik, a hangsúlyt a biztonságkritikus számítógéprendszerek szoftverjére helyezve. Először a számításba veendő szoftver hibákat ismerteti, majd ehhez kapcsolódóan a verifikáció és validáció feladatait. Ezt követően egy általános leképezési séma kerül ismertetésre, amely egy adott szoftver bemeneti és kimeneti tartománya közötti egy-egy értelmű kapcsolat leírására szolgál. Erre a sémára egy tesztmodell épül, amely tartalmazza a különböző hibaosztályokhoz tartozó tesztinputokat.

A közölt tesztmodell magában foglalja mind a verifikációs, mind pedig a validációs folyamatokat. A modell jelentősége abban áll, hogy megkönnyíti a világos megkülönböztetést a verifikációs és validációs tesztek között. Mindez a tesztek megtervezésének és kiértékelésének folyamatában bizonyul fontosnak és hasznosnak, mivel a két tesztelési feladat egymástól lényegesen eltérő megközelítést tesz szükségessé.

A cikk befejező része azt az esetet vizsgálja, amikor a szoftvert formális módszerek alkalmazásával tervezték meg. Ebben a részben tárgyalásra kerülnek a formális módszerek alkalmazásának következményei és problémái, valamint a módszereknek a verifikációra és a validációra való hatása.

1. Bevezetés

Ismeretes, hogy a komplex szoftverrendszerek megbízható üzemeltetése, felhasználása szigorú követelményeket támaszt a fejlesztésükre vonatkozóan. Ebbe szorosan beletartozik az a minőségbiztosítási technológia, amely a teljes fejlesztési folyamatot végigköveti, a specifikáció megadásától a kész rendszer üzembe helyezéséig [1-3]. A minőségi és megbízhatósági követelmények kielégítése szükségessé teszi azt, hogy a szoftvert úgynevezett *verifikációs* és *validációs* eljárásoknak vessük alá [1-7]. A verifikációban az egyes fejlesztési fázisok közötti összhang ellenőrzése a feladat, míg a validációval a végső rendszer ellenőrzésére kerül sor, annak eldöntésére, hogy az mennyire felel meg a felhasználó által előírt követelményeknek.

A verifikálási-validálási tevékenység pontos végrehajtása a következő főbb előnyökkel jár:

- Segít annak eldöntésében, hogy elkezdhetjük-e a fejlesztés soron következő fázisát.
- A fejlesztési folyamat korai szakaszában mutathat ki problémákat, hibákat.
- A szoftver minőségére, megbízhatóságára vonatkozóan mindvégig támpontokat, adatokat szolgáltat.
- Kimutathatja már korán azt is, hogy a szoftver nem teljesíti a követelményeket.

Mindez a szoftver előre megtervezett ellenőrzésével, intenzív tesztelésével kell hogy járjon az egyes fázisokban. Jelen közlemény olyan módszerekkel foglakozik, amelyek a szoftver verifikálásával és validálásával kapcsolatosak, ahol a tesztelés mindkét tevékenység integráns része. Itt a hangsúlyt az úgynevezett *biztonságkritikus számítógéprendszerekre* helyezzük [3,7-9], ahol a legfontosabb kritérium a biztonságos működés, az élet veszélyeztetése nélkül, valamint nagyobb természeti vagy anyagi kár okozása nélkül. Az ilyen típusú számító rendszert azért vettük itt számításba, mivel ez igényli a legalaposabban tervezett és végrehajtott eljárásokat, más egyéb rendszerekkel összehasonlítva.

A cikk egy tesztmodellt mutat be, amely egy leképezési sémán alapul. A séma a bemeneti és kimeneti tartományok közötti egy-egy értelmű leképezést írja le, egy adott szoftverrendszerre vonatkozóan. Ebbe a tesztbemenetek és a hibaosztályok szintén beletartoznak. A tesztmodell mind a verifikációs, mind pedig a validációs sémákat is magában foglalja. A modell jelentősége abban áll, hogy megkönnyíti a tiszta különbségtételt a verifikációs és a validációs tesztek között, ami fontos és hasznos a teszttervezés és a tesztkiértékelés során. Végezetül a *formális módszerek* szoftvertervezési felhasználásának következményeivel és problémáival foglakozik a cikk.

2. Hibamodell és alapfogalmak

A szoftverhibák alapvető sajátsága, hogy a működés teljes időtartama alatt jelen vannak. A kérdés az, hogy a hatásuk miként nyilvánul meg a különböző szituációkban. A hibáknak két alaposztályát különböztetjük meg:

a) Specifikációs hibák: Azok a hibák, amelyek a fejlesztési ciklus kezdetén képződnek, és a szoftver téves működésében nyilvánulnak meg azáltal, hogy nem teljesülnek a valós felhasználói követelmények. A téves

működés tág értelemben tekintendő: Egyaránt következménye lehet a hibás, a hiányos, valamint a következetlen, ellentmondást tartalmazó specifikálásnak. Ezt a kategóriát nevezhetjük még *külső* vagy *felhasználói hibának* is.

b) Programozási hibák: Azoknak a hibáknak a széles köre tartozik ide, amelyeket a programozók követnek el, az előzőleg már specifikált szoftver tervezési és kódolási folyamatában. Ennek a kategóriának másik szinonímái: *belső* vagy *fejlesztési hibák*. Néhány lehetséges hibatípust ezekből az alábbiakban sorolunk fel:

– hibás funkcióteljesítés,
– hiányzó funkciók,
– adatkezelési hibák az adatbázis elérése során,
– kezdési és befejezési hibák,
– hibák a felhasználói interfészben,
– határértékek alá vagy fölé kerülés,
– kódolási hiba,
– algoritmikus hiba,
– inicializálási hiba,
– a vezérlési folyamat hibája,
– adatátviteli hiba,
– input-output hiba,
– programblokkok közötti versenyhelyzet,
– programterhelési hiba.

A kiválasztott hibamodell nagymértékben meghatározza azokat a ráfordításokat, amelyeket a tesztelési folyamatok megtervezésében és végrehajtásában kell kifejteni [10-13].

A legfontosabb mérlegelési szempontok:

- Lehetőségek a teszttervezés megvalósítására, a költségek figyelembevételével.
- Előzetes ismeretek azokról a hibajelenségekről, melyek az adott fejlesztési technikához kapcsolódnak.
- A rendelkezésre álló hardver- és szoftvereszközök szolgáltatási köre és teljesítménye.

A gyakorlati tapasztalatok alapján állítható, hogy a teljes fejlesztési költségek mintegy 50%-át teszik ki a tesztelési költségek. Ez magában foglalja tesztelési folyamatok megtervezését és végrehajtását is. A szoftver-technológia fejlődésével a rendszerek mérete és bonyolultsága egyre növekszik. Ennek következményeként állandó igény mutatkozik arra, hogy újabb és hatékonyabb tesztelési módszereket és eszközöket dolgozzunk ki.

A szoftverfejlesztésben fontos követelmény a teljes folyamat konzisztens módon való végigvitele. Az egyes fejlesztési állomások eredményei saját megjelenési formával rendelkeznek. A folyamat helyes végigvitele megköveteli, hogy az egyes reprezentációk közötti összhang bizonyítását. Végül is, azt kell bizonyítani, hogy a végső szoftver termék száz százalékig megfelel a kiindulási specifikációnak. Ennek a bizonyítási folyamatnak a megvalósítása oly módon történik, hogy az egymást közvetlenül követő fejlesztési fázisok közötti összhangot bizonyítjuk lépésenként. Ha egy fázisnál diszkrepancia mutatkozik, akkor azt addig kell módosítani, amíg az nem harmonizál az előző fázissal. A leírt tevékenységet, amelyben két egymást követő fázis közötti ekvivalenciát bizonyítjuk, verifikációnak nevezzük. Ennek pontosabb definíciója a következő:

Verifikáció: Az a folyamat, amelyben igazoljuk, hogy a szoftver egy fejlesztési fázisban teljesíti mindazokat a követelményeket, amelyeket az előző fázisban specifikáltunk.

A lépésenkénti verifikálás elvileg elegendő az ekvivalencia igazolására a kiindulási és a befejező fázis között. Mindazonáltal, mivel a teljes folyamat általában nem egzakt, vagyis nem biztosít abszolút bizonyítást egyik lépésben sem, egy külön önálló *végső verifikációra* is szükség van, amit a specifikáció és a végtermék között hajtunk végre.

Másfelől nézve, ennél a pontnál meg kell jegyeznünk, hogy a verifikációs folyamat tökéletes megvalósítása sem garantálná a végtermék tökéletes használhatóságát. Mindaz, amit garantálni lehet, a kiindulási specifikációval való ekvivalencia teljesülése. Abban az esetben, ha hiba vagy tökéletlenség volt a specifikációban, a végtermék nem fogja kielégíteni a felhasználói követelményeket. Emiatt szükség van még egy külön vizsgálati folyamatra is, amiben a terméket az eredeti rendeltetése szempontjából ellenőrizzük. Az ilyen jellegű bizonyítási folyamatot *validációnak* nevezzük. Ennek definíciója a következő:

Validáció: A szoftvernek olyan vizsgálata és kiértékelése, amiben meghatározzuk, hogy minden szempontból teljesíti-e a felhasználói követelményeket.

Egy biztonságkritikus rendszer esetben a következőket kell bizonyítani:

– funkcionálisan megfelelő működés,
– megfelelő teljesítmény,
– a biztonsági követelmények kielégítése.

A tökéletlen specifikálás következményeként leginkább a biztonság lesz veszélyeztetve. Mindezek után, a végső validációs eljárásban azt kell eldönteni, hogy a teljes rendszer biztonságos-e vagy sem. Ha valamilyen probléma adódik, akkor vissza kell térni a kezdeti specifikációhoz, és módosítani kell azt. A módosítás azzal jár, hogy újra kell tervezni a rendszert, a szükséges változtatások végigvitelével, minden egyes verifikációs fázist elvégezve, másrészt új validálásra is sort kell keríteni.

A verifikáció és validáció együtt kezelendő, teljes összhangban. Ezt a tényt a széles körben elterjedt *„V&V" eljárás* elnevezés is kifejezi [3]. A két fogalmat azonban néha összekeverik, annak ellenére, hogy lényegesen eltérő tevékenységeket jelölnek. A verifikáció annak ellenőrzése, hogy a program megfelel-e a specifikációjának. A validáció pedig annak eldöntésére irányul, hogy a megvalósított program teljesíti-e a felhasználó elvárásait.

A két tesztelési feladat egymástól lényegesen eltérő megközelítést igényel. A verifikáció esetében a teszteknek az előállított, rendelkezésre álló megjelenési formák közötti ekvivalenciát kell igazolniuk. Ehhez számos jól bevált teszttervezési módszert tudunk felhasználni [1,2,10-12]. A validációs tesztek előállításakor viszont arra kell törekedni, hogy egy biztonságkritikus rendszert többféle komplikált működési helyzetbe hozzunk, ellenőrzendő, hogy az mindegyik helyzetben ki-

állja-e a próbát, és nem okozhat károkat [3,7,8]. Az ilyen tesztek képzéséhez a felhasználási cél, valamint a biztonságosság elérése ad útmutatót.

3. Verifikációs és validációs modell

Amint már deklaráltuk, a szoftverhibák a programozás során alakulnak ki, illetve hibás vagy nem teljes specifikációból származnak. Az első kategória a nem megfelelő fejlesztési folyamat következménye, a második pedig a végső felhasználási követelményekkel való összhang hiánya. A következőkben ezt a két kategóriát rendre *fejlesztési hibának,* illetve *felhasználói hibának* fogjuk nevezni.

A szoftverhibák a program helytelen működésében nyilvánulnak meg, amikor a hibás kódszegmens végrehajtására kerül sor, annak a bemeneti adathalmaznak a hatására, amely előhozza az adott hibát. Ugyanakkor a kód többi része már jól működhet más bemenetekre nézve. Egy szoftverrendszert olyan egységnek tekinthetünk, ami *egy bemeneti halmazt egy kimeneti halmazra képez le.* Egy rendszernek nagyon sok bemeneti értéke lehetséges. Az egyszerűség kedvéért a bemeneti értékek kombinációját és szekvenciáját egyetlen bemeneti elemnek fogjuk tekinteni. A bemenetei értékekre adott válaszértékek képezik a kimeneti értékek halmazát.

Legyen a szoftver összes lehetséges bemeneti értékének halmaza **INPD** (input domain, – bemeneti tartomány), és az összes lehetséges kimeneti válasz halmaza **OUTD** (output domain, – kimeneti tartomány). Ezzel az INPD-nek a szoftver által történő leképezését az OUTD-re a következő formában fogjuk definiálni:

SWM (INPD) = OUTD. (1)

Az (1) reláció azt jelenti, hogy a rendszer működési tulajdonságai, vagyis az SWM leképezés (software mapping), határozzák meg az INPD és az OUTD elemei közötti megfelelést. A relációt ebben a formájában érvényesnek fogjuk tekinteni a szoftverfejlesztési ciklus bármelyik fázisában.

Jelöljük továbbá **INER**-rel (input errors) azon bemeneti értékek halmazát, amelyek hibás működést okoznak, míg a hibás válaszértékek halmaza legyen **OUTER** (output errors). Ezekre a halmazokra a következő leképezési reláció áll fenn:

SWM (INER) = OUTER. (2)

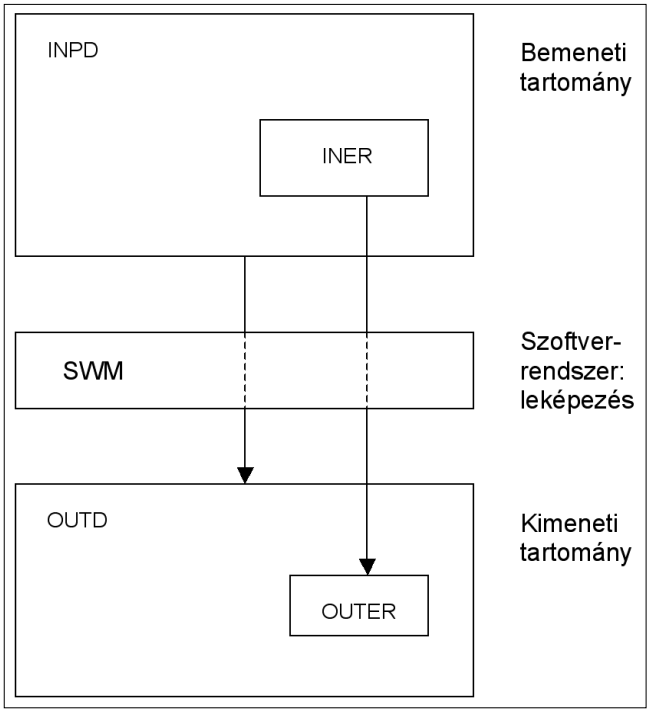
A fenti két leképezés sémáját az *1. ábra* mutatja be.

Ha a leképezési modellünkbe bevonjuk a hibafelfedő teszteket, akkor ez az INER és OUTER halmazok módosulását fogja eredményezni. Tegyük fel, hogy egy INERT elnevezésű teszthalmaz (input-error tests) alkalmas arra, hogy felfedje a még felderítetlen hibák egy részhalmazát. Ebben az esetben az INERT és INER szükségszerűen közös elemekkel kell hogy rendelkezzenek. Az elvárható következmény ekkor a felfedett hibák eltávolítása. Ez azt jelenti, hogy a javított szoftver egy módosított kimeneti tartományt fog produkálni, egy olyan OUTER részhalmazzal, amely már nem képviseli tovább a detektált és ily módon eltávolított hibákat. Ezek után a javított szoftverre vonatkozó új leképezési reláció

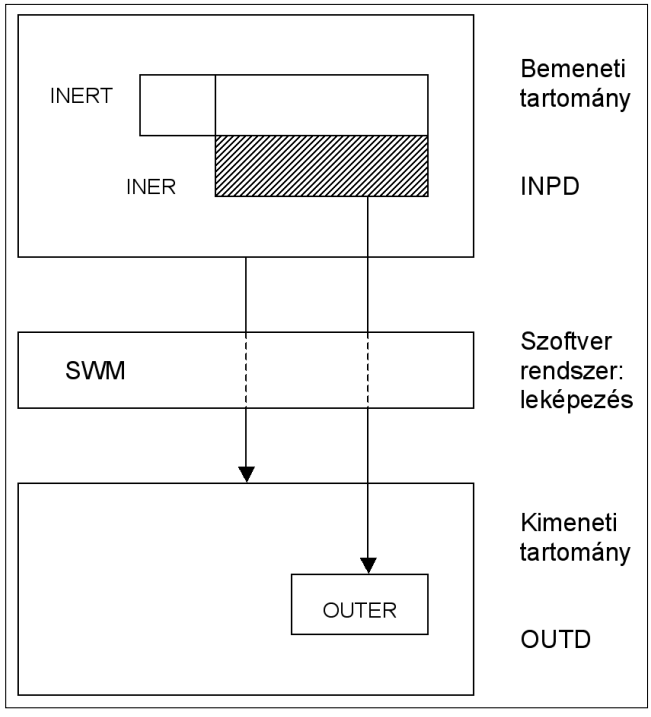
SWM (INER – INERT) = OUTER (3)

lesz, ahol a mínusz jel a halmazok közötti kivonást képviseli. Az ennek megfelelő leképezési séma a *2. ábrán* látható.

1. ábra Szoftver-leképezési séma hibák esetén



2. ábra Leképezési séma tesztelés után



Ennél a pontnál már rátérhetünk a verifikáció és validáció kérdésének vizsgálatára. Itt a következő jelöléseket vezetjük be:

- A fejlesztési hibákban megnyilvánuló bemenetek halmaza: **DEFI** (development-fault inputs). Ennek a halmaznak a kimeneti leképezése **DEFO** (development-fault outputs).
- A felhasználói hibákban megnyilvánuló bemenetek halmaza: **USFI** (user-fault inputs). Ennek a halmaznak a kimeneti leképezése **USFO** (user-fault outputs).
- A fejlesztési hibák detektálására készített tesztbemenetek halmaza, azaz a verifikációs tesztek halmaza: **VERT** (verification tests).
- A felhasználói hibák detektálására készített tesztbemenetek halmaza, azaz a validációs tesztek halmaza: **VALT** (validation tests).

Miután megvan az eljárásunk arra, hogy különböző input halmazokat output halmazokra képezzünk le, általános modellt tudunk adni a verifikáció és validáció leképezési sémájára, egy adott szoftverrendszerre vonatkozóan. A leképezési relációk a következők lesznek:

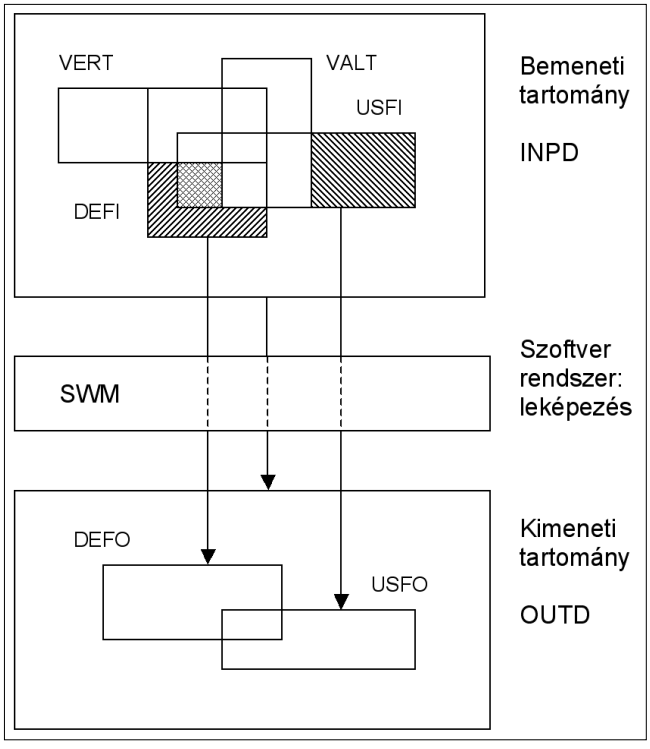
$$\mathbf{SWM\ (INPD) = OUTD.} \quad (4)$$

$$\mathbf{SWM\ (DEFI - (VERT \cup VALT)) = DEFO.} \quad (5)$$

$$\mathbf{SWM\ (USFI - (VERT \cup VALT)) = USFO.} \quad (6)$$

A fentiekben az (5) reláció a verifikációs leképezést, míg a (6) reláció a validációs leképezést fejezi ki. Ezekből a relációkból látható, hogy (5) a felfedetlen fejlesztési hibákat, a (6) pedig a felfedetlen felhasználói hibákat képviseli. A (4), (5) és (6) relációkat a *3. ábra* összetett leképezési sémája mutatja be.

3. ábra
Leképezési séma verifikációval és validációval



4. Formális módszerek használata

A formális módszerek matematikai technikát alkalmaznak a számítógépek hardverjének és szoftverjének specifikálásában, tervezésében és analízisében [7,14-15]. Ismeretes, hogy a biztonságkritikus rendszerek fejlesztésével kapcsolatos problémák egy jó része a specifikációs hiányosságokból ered [3]. A specifikációnak egyértelműnek, teljesnek, konzisztensnek és helyesnek kell lennie. Azok a dokumentumok, amelyek természetes nyelven íródtak, mindig ki vannak téve a félreértésnek. Ugyancsak nehéz elérni azt is, hogy ezek a dokumentumok a tervezendő rendszer teljes és helyes leírását képviseljék, vagy még azt is kimutatni, hogy ezek konzisztensek egymással.

A formális módszerek *formális nyelvek* használatán alapulnak, amelyeknek precíz és szigorú szabályaik vannak. Ez a sajátság lehetővé teszi, hogy a specifikálást olyan módon készítsük el, amely egyértelműen interpretálható. Mindemellett még az is lehetővé válik, hogy automatizáltan ellenőrizzük a specifikációt, azzal a céllal, hogy kihagyásokat és következetlenségeket találjunk benne, vagyis hogy bizonyítsuk a teljességet és a konzisztenciát.

Azok a nyelvek, amelyek ezt a célt szolgálják, a *rendszer-specifikációs nyelv,* vagy *formális specifikációs nyelv* elnevezést viselik. Használatuk számos potenciális előnyt kínál a fejlesztési ciklus mindegyik fázisában [7,14-21].

Az egyik legnagyobb előny az, hogy automatikus tesztek hajthatók végre az ilyen leírás alapján. Ez lehetővé teszi, hogy szoftver eszközökkel ellenőrizzünk bizonyos hibaosztályokat, másrészt hogy különböző leírásokat hasonlítsunk össze annak eldöntésére, hogy ekvivalensek-e. A terv különböző fázisai ugyanannak a rendszernek a leírásai, s így ezeknek funkcionálisan ekvivalensnek kell lenniük. Ha minden egyes reprezentáció megfelelő formában készült el, akkor közöttük egyenként bizonyítható lesz az ekvivalencia.

Mint látható, ez a folyamat nem más mint maga a verifikálás. A *4. ábra* azt mutatja, hogy minden egyes transzformációt annak ellenőrzése követ, hogy az helyesen hajtódott-e végre. Ez annak kimutatását jelenti, hogy egy adott fázis bemenetét adó leírás funkcionálisan ekvivalens azzal, ami a fázis kimenetén állt elő.

Ideális esetben a fent vázolt folyamat transzformációs eljárásai teljes mértékben automatizálva vannak, emberi beavatkozás nélkül, és mindegyik fázisban hibamentesen hajtódnak végre. Ha ez teljesül, akkor a kívülről származó verifikációs tesztsorozatok teljesen elhagyhatók lesznek. A tesztmodellünkben ez azzal jár, hogy

$$\mathbf{DEFI} = \varnothing, \qquad \mathbf{VERT} = \varnothing,$$

ahol a $\varnothing$ szimbólum az üres halmazt jelöli. Ha ezt a két eredményt az (5) relációba helyettesítjük, akkor

$$\mathbf{SWM}\ (\varnothing - (\varnothing \cup \mathbf{VALT})) = \mathbf{SWM}\ (\varnothing) = \mathbf{DEFO} = \varnothing \quad (7)$$

adódik.

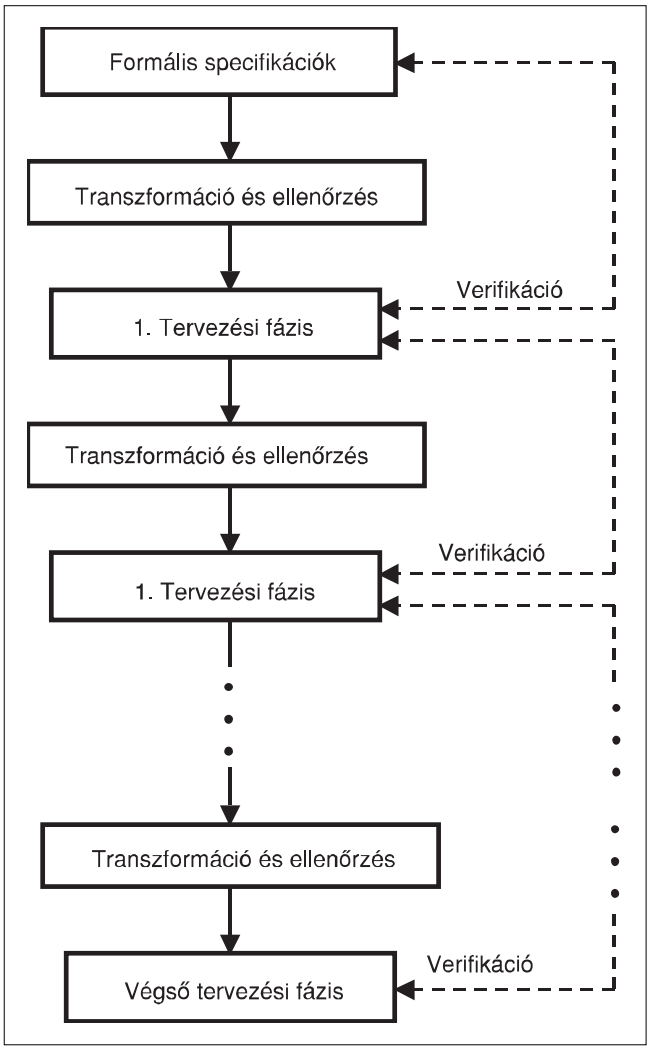
Másrészről, a szoftver kezdeti formális specifikálása mindig kézi úton történik, így emiatt a tervezési hibák sosem zárhatóak ki itt, még akkor sem, ha automatizált konzisztencia-ellenőrzés áll rendelkezésre. Ennek az oka az, hogy egy konzisztens terv még önmagában véve nem garantálja a felhasználói követelmények maradéktalan teljesítését.

Hasonlóképpen, nincsen garancia a biztonsági követelmények teljes mértékű kielégítésére sem. Következésképpen a külső validációs tesztelés alkalmazására mindig szükség van. A fenti ideális eset figyelembevételével a (6) reláció redukált alakja a következő lesz (8):

SWM (USFI – (∅ ∪ VALT)) = SWM (USFI – VALT) = USFO.

Végezetül hangsúlyoznunk kell, hogy a vázolt eljárás nem tekinthető csodaszernek. Jóllehet a formális módszerek használata számos előnnyel jár, jelentős számú megszorítást is hordoz magában a fejlesztési módszerekre nézve. Mivel egy lánc erősségét a leggyengébb eleme határozza meg, a maximális haszon elérése érdekében szükségessé válik a formális bizonyítások elvégzése a fejlesztés minden egyes fázisában. Ez komoly kihatásokkal van a projekten belül alkalmazható tervezési módszerekre. A legtöbb esetben a teljesen automatizált végrehajtás nem alkalmazható, így továbbra is szükség van a jól képzett tervezők szoros együttműködésére és beavatkozására. Jelenleg a formális módszerek legfőbb előnye abban van, hogy a tervezési és verifikálási folyamatok nagyobb megbízhatóságú végrehajtását teszik lehetővé. A következőkben néhány konkrét problémát sorolunk fel a formális módszerekkel kapcsolatosan:

4. ábra Formális módszerek alkalmazásának folyamata



1) A gyakorlati alkalmazások területén eddig még kevés tapasztalat gyűlt össze. A fő gondot a számítási komplexitás jelenti, amit igen nehéz előzetesen meghatározni. A komplexitásra vonatkozóan az tapasztalható, hogy egyes algoritmusok az NP-teljes feladatok osztályába tartoznak. Mint ismeretes, az NP-teljes feladatok számítási komplexitása olyan, amire várhatóan nem létezik felülről korlátozó véges fokszámú polinom, ahol a polinom változója a feladat méretét képviseli. Ez valójában azt jelenti, hogy a számítási lépések száma véges, de megjósolhatatlanul nagy.

2) A biztonságkritikus rendszerek biztonságigazolási folyamata, vagyis a validációs folyamat elvileg nem automatizálható teljes mértékben. Ez azért van így, mert ebben az esetben a tesztelésnek mindig kell hogy legyenek olyan elemei, amelyek kívül esnek a rendszerspecifikáción, vagyis ezek a kiegészítő elemek függetlenek a specifikációtól. Az alapvető ok az, hogy azok a biztonsági problémák, amelyek a hiányos vagy téves specifikációból erednek, egyedül csak ezen a módon fedezhetők fel. Természetesen ez az állítás nem csak a klasszikus módszerekre vonatkozik, hanem a formálisokra is.

3) Jelenleg nem áll rendelkezésre olyan egzakt formális specifikálási módszer, amellyel magának a biztonságnak az elvét tudnánk számításba venni. Más szóval, a biztonsági célokat közvetlenül szolgáló formális specifikációs módszerek még nincsenek kidolgozva.

A jelenlegi helyzet és az egyre fokozódó igény további jelentős kutatási és fejlesztési erőfeszítéseket követel, annak érdekében, hogy a formális módszerek hatékonyabbá váljanak a gyakorlatban. Hogy jelezzük az irányt, befejezésül néhány nyitott problémát, illetve kérdést vetünk fel az alábbiakban:

- Lehetséges-e formalizálni a biztonság szintet, amit el kell érni a tervezési folyamatban?
- Ha igen, ez milyen módon valósítható meg?
- Lehetséges-e meghatározni egy hibatűrő rendszer biztonságát kvantitatív módon?
- Lehetséges-e közvetlen kapcsolatot megteremteni egy hibatűrő struktúra és a formális tervezés között?

5. Befejező megállapítások

Ez a közlemény a szoftvertesztelésre vonatkozóan egy általános modellre ad javaslatot. A modell egy leképezési sémán alapszik, amely a bemeneti és kimeneti tartományokat foglalja magában, valamint a különböző tesztelési feltételeket és hibalehetőségeket.

A bemutatott elv fő célja az, hogy világosan megkülönböztesse a verifikációra és a validációra szolgáló teszteket, ami különösen fontos és hasznos a biztonságkritikus rendszerek esetében. A két tesztelési feladat ugyanis egymástól lényegesen eltérő megközelítést igényel. Mint láthattuk, az itt bemutatott modell alkalmazható a formális módszereken alapuló szoftverfejlesztésnél is. Ugyanakkor a cikk kimutatta azt is, hogy a validációs eljárások velejáró elméleti korlátozásokkal bírnak, amikor formális specifikációt alkalmazunk.

A biztonság-orientált informatikai rendszerek tervezésében ma már döntő szerepet játszik a szoftver eszközök felhasználása. A szoftver meghatározó szerepe a rendszerek üzemeltetése során is érvényesül, a bennük megvalósított komponenseken keresztül. Mindezek miatt egyre inkább növekszik a megbízható szoftver előállítására fordítandó kutatási-fejlesztési tevékenységek fontossága.

Végezetül, mint ismeretes, az informatikai rendszerek hardver összetevője szintén szigorú és pontos fejlesztési és gyártási technológiát igényel. Ez az irányelv mindenek előtt a biztonságkritikus rendszerekre érvényes, ahol a hardver és szoftver együttes tervezése széles körben használatos megközelítés [3,5].

Ebben a megközelítésben a végleges megosztás a két szféra funkciói között a tervezési folyamat során dől el. Egy hardver rendszer biztonságos és megbízható működése szintén megköveteli a verifikálást és a validálást, mind a tervezési, mind pedig a gyártási ciklusban [16].

Ami a fentiekben bemutatott tesztmodellt illeti, belátható, hogy az, bizonyos pótlólagos megfontolásokkal, kiterjeszthető a hardver rendszerek kezelésére is. Egy ilyen jellegű kiterjesztés hibamodellje a hardver tervezési hibáit, gyártási hibáit, valamint üzemeltetési hibáit foglalja magában.

Irodalom

[1] Ian Sommerville:
Software Engineering, 6th Ed.,
Addison-Wesley Publ. Company, Inc., USA, 2001.

[2] Roger S. Pressman:
Software Engineering, 5th Ed.,
McGraw-Hill Book Company, USA, 2001.

[3] Neil Storey: Safety-Critical Computer Systems,
Addison-Wesley-Longman, Inc., New York, 1996.

[4] Marvin V. Zelkowitz:
Role of Verification in the Software Specification Process,
Advances in Computers,
(Editor: Marshall C. Yovits), Vol. 36, pp.43–109.,
Academic Press, Inc., San Diego, 1993.

[5] Jean-Michel Bergé, Oz Levia, Jacques Rouillard:
Hardware/Software Co-Design and Co-Verification,
Kluwer Academic Publishers, Dordrecht, NL, 1997.

[6] Nicolas Halbwachs, Doron Peled (Editors):
Computer Aided Verification,
Lecture Notes in Computer Science, Vol. 1633,
Springer-Verlag, Berlin, 1999.

[7] József Sziray, Balázs Benyó, István Majzik,
András Pataricza, Júlia Góth, Levente Kalotai,
Tamás Heckenast: Quality Assurance and
Verification of Software Systems, (In Hungarian),
Széchenyi College, Budapest Technical and
Economical University, University of Veszprém, 2000.

[8] Nancy G. Leveson:
Safeware: System Safety and Computers,
Addison-Wesley Publ. Company Inc., USA, 1995.

[9] Dhiraj K. Pradhan:
Fault-Tolerant Computer System Design,
Prentice-Hall, Inc., USA, 1996.

[10] Glenford J. Myers:
The Art of Software Testing,
John Wiley & Sons, Inc., New York, 1979.

[11] Cem Kaner, Jack Falk, Hung Quoc Nguyen:
Testing Computer Software,
Van Nostrand Reinhold, Inc., New York, 1993.

[12] Boris Beizer:
Black-Box Testing, Techniques for
Functional Testing of Software and Systems,
John Wiley & Sons, Inc., New York, 1995.

[13] Péter Várady, Balázs Benyó:
A Systematic Method for the Behavioural Test and
Analysis of Embedded Systems, IEEE International
Conf. on Intelligent Engineering Systems, Proc.,
pp.177–180., Portoroz, Slovenia, Sept. 17-19, 2000.

[14] Michael J. C. Gordon:
Programming Language Theory and its Implementation,
Prentice Hall International Ltd, Great Britain, 1988.

[15] Constance Heitmeyer, Dino Mandrioli (Editors):
Formal Methods for Real-Time Computing,
John Wiley & Sons, Inc., New York, 1996.

[16] Balázs Benyó, József Sziray:
The Use of VHDL Models for Design Verification,
IEEE European Test Workshop, Proc.,
pp. 289–290., Cascais, Portugal, May 23-26, 2000.

[17] Martin Fowler, Kendall Scott:
UML Distilled: Applying the Standard Object
Modeling Language,
Addison-Wesley-Longman, Inc., USA, 1997.

[18] Mark Priestley:
Practical Object-Oriented Design with UML,
McGraw-Hill Publishing Company, GB, 2000.

[19] Csaba Szász, József Sziray:
Run-Time Verification of UML State-Chart
Implementations, IEEE Intern. Conf. on Intelligent
Engineering Syst., Proc., pp.355–358.,
Portoroz, Slovenia, Sept. 17-19, 2000.

[20] Zsigmond Pap, István Majzik, András Pataricza,
András Szegi:
Completeness Analysis of UML Statechart Specific.,
IEEE Design and Diagnostics of Electronic Circuits
and Systems Workshop, Proc., pp.83–90,
Győr, Hungary, April 18-20, 2001.

[21] Eric J. Braude: Software Engineering,
An Object-Oriented Perspective,
John Wiley & Sons, Inc., New York, 2001.