MÉRÉSTECHNIKA ÉS INFORMÁCIÓS RENDSZEREK TANSZÉK

MODELL ALAPÚ AUTOMATIKUS TESZTGENERÁLÁS

DIPLOMATERV

Micskei Zoltán

Műszaki informatika szak

Konzulens: Dr. Majzik István

Budapest
2005. május

Modell alapú automatikus tesztgenerálás

Diplomaterv kiírás Micskei Zoltán műszaki informatika szakos hallgató részére

A szoftverfejlesztés során a tesztelés a fejlesztési folyamat sok időt és erőforrást igénylő feladata. A tesztelés első lépése a tesztek megtervezése, a megfelelő bemenet - elvárt kimenet sorozatok összeállítása. A szoftverfejlesztésre vonatkozó szabványok gyakran előírják a tesztelés során a forráskódra vonatkozó úgynevezett fedettségi kritériumok teljesítését (pl. minden utasítás, vagy minden döntési ág bejárását a tesztelés során). A kritériumoknak megfelelő, részletes és hatékony tesztkészlet előállítása a lehetséges bemeneti kombinációk nagy száma miatt nehéz feladat.

Modell alapú fejlesztés esetén lehetővé válik, hogy a tesztkészletek generálását automatikusan végezzük el. Elsősorban az egyszerű bemenő eseményeket és akciókat alkalmazó, eseményvezérelt beágyazott rendszerek esetén van erre lehetőség. A jelölt az Önálló laboratórium tárgy keretében megvizsgálta modell ellenőrző eszközök (SPIN és SMV) használatát tesztek automatikus generálásának céljaira, és elvégezte a modell ellenőrzők optimális beállításainak meghatározását. A diplomaterv célja egy egységes teszt generáló keretrendszer összeállítása és a még hiányzó funkciók beillesztése.

A diplomaterv kidolgozása a következő részfeladatok megoldását igényli:

- Tekintse át a modell alapú tesztgenerálás alapfeladatait és a jelenleg rendelkezésre álló megoldásokat.
- Hozzon létre egy egységes kezelői felületen elérhető környezetet, amely lehetővé teszi strukturális tesztek automatikus generálását UML állapottérkép diagramok alapján, állapot- és állapot-átmenet fedési kritériumok figyelembe vételével. Használja fel ehhez a SPIN modell ellenőrzőt. Valósítsa meg a tesztkészlet optimalizálását.
- Vizsgálja meg annak lehetőségét, hogy az időzítési specifikációt is tartalmazó UML állapottérképek alapján hogyan történhet a tesztek generálása! Ebből a célból vizsgálja meg az Uppaal modell ellenőrző képességeit!
- Vizsgálja meg annak lehetőségét, hogy a generált tesztek hogyan illeszthetők a Rational Robot teszt végrehajtó rendszerhez.
- Egy mintapéldán keresztül mutassa be a teszt generálási környezet működését.
- Végezzen mintakísérleteket annak demonstrálására, hogy a modell alapján generált tesztek milyen fedettséget érnek el a tényleges forráskódra vonatkoztatva (utasítás- illetve döntési ág fedettség). Vizsgáljon meg többféle, az UML állapottérkép modell alapján történő forráskód generáláshoz kidolgozott megvalósítási mintát is.

Budapest, 2005. január 30.

dr. Majzik István
tanszéki konzulens

Nyilatkozat

Alulírott, ***Micskei Zoltán***, a Budapesti Műszaki és Gazdaságtudományi Egyetem hallgatója kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, és a diplomatervben csak a megadott forrásokat használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

.....

Micskei Zoltán

Összefoglaló

A szoftverfejlesztés során a tesztelés mindig is a folyamat fontos feladata, mely sok időt és erőforrást igényel. A klasszikus tesztelési módszerek kezdeti lépése a tesztek megtervezése, a megfelelő bemenet – elvárt kimenet-sorozatok összeállítása. A szoftverfejlesztésre vonatkozó szabványok gyakran előírják a tesztelés során a forráskódra vonatkozó úgynevezett fedettségi kritériumok teljesítését (pl. minden utasítás, vagy minden döntési ág bejárását a tesztelés során). A kritériumoknak megfelelő, részletes és hatékony tesztkészlet előállítása a lehetséges bemeneti kombinációk nagy száma miatt nehéz feladat.

Modell alapú fejlesztés esetén lehetővé válik, hogy a tesztelés bizonyos fázisait automatikusan végezzük el. A diplomamunka első fejezete áttekinti a modellezés és ellenőrzés során használt technikákat.

A következő fejezet ismerteti a modell alapú tesztelés lehetőségeit és eddigi eredményeit, valamint bemutat egy kiválasztott technikát: a modell ellenőrző segítségével történő tesztgenerálást [HLSC01], ami elsősorban eseményvezérelt beágyazott rendszerek esetén lehet hatékony. Ennek lényege, hogy a teszt fedettségi kritériumot a modell ellenőrző számára, mint követelményt fogalmazzuk meg, ez a legtöbb eszköz esetén temporális logikai állításokkal történik. A követelmény rögzíti a tesztelési cél negáltját (pl. azt előírva, hogy egy adott utasítássorozat nem hajtható végre), ezt ellenőrizve a modell ellenőrző egy ellenpéldát talál (itt megmutatja azt a bemeneti sorozatot, amivel az utasítássorozat ténylegesen végrehajtható). Ez az ellenpélda már közvetlenül használható tesztsorozat konstruálására.

A diplomamunka során elkészült egy eszköz, mely a fenti módszert alkalmazza a gyakorlatban. A harmadik fejezet ismerteti a program környezetét, a felhasznált külső komponenseket és adatformátumokat, a negyedik pedig részletesen tárgyalja a megvalósított program tervezését és implementációját.

A tesztgenerálás igényei (minél rövidebb és hatékonyabb tesztek összeállítása) sok esetben eltérnek a klasszikus modell ellenőrzés igényeitől (biztonsági és élőségi feltételek ellenőrzése). Igen fontos tehát kidolgozni a modell ellenőrzőnek azon beállításait, amelyekkel a tesztek hatékonysága mellett a rövid futási idő és a mérsékelt erőforrásigény is biztosítható. A diplomaterv ötödik fejezete egy mintapéldán végzett méréseken keresztül bemutatja a SPIN és az SMV modell ellenőrző optimalizációs lehetőségeit és a kapcsolódó futási eredményeket. Végül demonstrálja a tesztgenerálás alkalmazhatóságát egy valós ipari rendszer modelljén.

A program által generált tesztek a modellen értelmezhetőek, így fontos kérdés megvizsgálni, hogy miképpen lehet ezeket felhasználni a modellhez elkészített implementáció teszteléséhez. A hatodik fejezet bemutat egy olyan módszert, melynek segítségével könnyedén illeszthető teszt futtató környezetekhez (JUnit, Rational Robot) az előzőleg generált tesztek. A fejezetet az implementáció kódjának lefedettségét vizsgáló mérések zárják.

Végül a hetedik fejezetben a diplomamunka tárgyalja azokat a kiterjesztéseket is, amelyek szükségesek valósidejű rendszerek (időzítési kritériumokat is tartalmazó) tesztsorozatainak generálásához.

Overview

Testing is an essential, but time and resource consuming activity in the software development process. The related standard often specify code coverage (statement or branch) levels to met. Generating a short, but effective test suite usually needs a lot of manual work and expert knowledge. In a model-based process, among other subtasks, test construction and test execution can also be partially automated. The first part of the thesis summarizes the techniques used to model and test embedded systems. After that it reviews the recent publications on model-based testing and the various commercial and academic tools used in automatic test generation.

Based on the method suggested in [HLSC01], the thesis presents a test generator tool, which can be used for test set generation in the development process of event-driven embedded systems. For a selected test coverage criterion and from the UML statechart model of the system the program generates test cases using the SPIN model checker. The test generator tool supports currently the test sequence construction on the basis of the "all states", "all transitions" and "custom formula" coverage criteria. The necessary model transformation [LMM99] and requirement generation steps are performed automatically.

The main steps of the test generator are:

1. Transform the UML model into the model checker's input format.
2. Generate the test requirements satisfying the given test goal (coverage criterion) and formalize them in temporal logic.
3. For each of the formula check it's negated version, the generated counter-example will be the test for the actual formula.
4. Extract the input and output events from the trail files of the model checker and construct the desired test cases from them.

The environment and components used by the program are described in the fourth chapter. It outlines the whole test generation process. Chapter four contains the implementation details, the logical and physical model of the application.

The configuration of the model checker in the case of test generation, namely the settings required for constructing a short and minimal test suite, differs from the usual needs of classic model checking problems. The thesis analyzes the possible settings of the model checker SPIN by measuring the efficiency of test construction in the case of different real-life statechart models, and introduces an optimized setting for test generation.

The generated test cases operate on the statechart model, an important task is to produce tests for an implementation of the model. Chapter six describes a method to transform the test in a format so that they can be run in popular test execution systems (JUnit, Rational Robot). The code coverage of the generated tests was measured and the results are presented at the end of the chapter.

The test generator can be extended for real-time applications, in this case the model is available in the form of timed automata, and the model checker to be used is Uppaal [UPPAAL]. Chapter seven demonstrates it with an example, and analyses the problem to be solved, if the test generator program will be ported to Uppaal.

Tartalomjegyzék

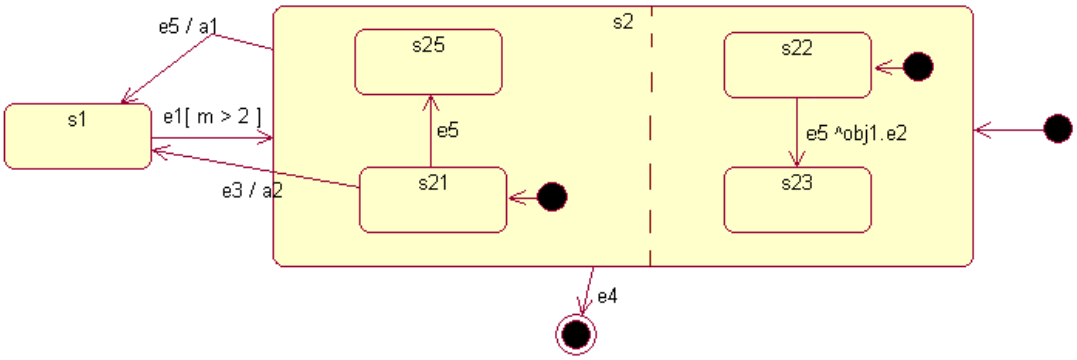
1 Beágyazott rendszerek modellezése és tesztelése

Manapság –az élet legváltozatosabb területein megjelenő– bonyolultabbnál bonyolultabb mikroprocesszoros eszközöknek köszönhetően egyre hangsúlyosabb lesz a *beágyazott rendszerek* fejlesztése és alapos tesztelése. Az ilyen rendszerek közös jellemzői, hogy főleg *vezérlés-orientáltak*, lényegi funkcióinak leírása könnyen megadható a külvilágtól származó esemény és a rá adott válasz segítségével. Egyúttal kevés adattal dolgoznak, s a rendszer leírása során nem ezek dominálnak. Mindezek miatt működésükre nagyon jól használhatók a különböző állapotgépes és állapottérképes leírások.

1.1 Modellezés állapottérképekkel

Az állapot alapú modellezés legegyszerűbb módja az egyszerű állapotgépek, illetve a leírásukra használt *állapot-diagramok* használata, azonban ezek alkalmazása gyakorlati alkalmazásokban igen kényelmetlen. Párhuzamos végrehajtású, átlapolt műveletek ábrázolása automata részek direktszorzatával oldható meg, ez pedig nagyon megnöveli az állapotteret. Az iteratív tervezési módszereket és a könnyebb áttekintést segítő hierarchikus kezelést pedig szintén nélkülöznünk kell.

A fent vázolt problémákat oldják meg az *állapottérképek* a hiányolt konstrukciók bevezetésével. Több dialektusuk is létezik, a diplomaterv további részében az UML szabványban [UML03] szereplő változatukat használjuk. A továbbiakban egy egyszerű példán keresztül szemléltetjük az állapottérkép diagramok elemkészletének leggyakrabban használt tagjait!



1. ábra: UML állapottérkép diagram

Az állapottérkép alapeleme az *állapot*, melynek jele egy lekerekített sarkú téglalap, az állapotot neve azonosítja (pl. s1, s23). A kezdő és végállapotnak speciális jele van: ● illetve ◉. Az állapokat tovább lehet finomítani, azaz felbontani újabb al-állapotokra, és így eggyel növelni a hierarchiaszintet. A hierarchia lehet OR típusú, ilyenkor az adott állapotot állapotok szekvenciájára (tulajdonképpen egy sorrendi automatára) bontjuk tovább, amely állapotok közül mindig csupán egy lehet aktív. AND típusú finomításra látunk példát s2 esetén, ahol két párhuzamosan működő régióra bontjuk szét az állapotot, ez esetben szaggatott vonallal választjuk el a két részt. Ha a

rendszer s2 állapotban tartózkodik, akkor minden egyes al-régiójában kötelező aktív állapotnak lennie.
Az állapotok között átvezető nyilakat *átmeneteknek* (tranzícióknak) nevezzük. Egy állapot címkéje a következő elemekből állhat:

```
trigger [őrfeltétel] / akció_szekvencia
```

A trigger az az esemény, melynek bekövetkezése szükséges az átmenet végrehajtásához (az ábrán ilyen az e1, e3, stb.). Ha egy állapot trigger eseménye bekövetkezett, és az őrfeltétele is teljesült, akkor arra az állapotra azt mondjuk, hogy engedélyezett, ki lehet választani tüzelésre. Ha egy átmenet tüzel, akkor végrehajtódik az akciószekvenciája, ez lehet akció végrehajtása (a példában a1, a2), valamely változó értékének módosítása, vagy esemény küldése egy másik objektumnak (erre látunk példát az s22-ből s23-ba lépő átmenetnél). Az UML állapottérképeknél megengedett a hierarchiaszintek közötti átmenet is (s21-ből s1-be vezető).

Az állapottérképek informális működési szemantikája a következőképp írható le: Indulás után az állapottérképet megvalósító állapotgép a legfelső szintű kezdőállapotnak megfelelő állapotba kerül, automatikusan az válik aktívvá. Ha ez összetett állapot, azaz tovább finomított, akkor a hozzá tartozó alrégió(k)ban is belép a kezdőállapot(ok) által jelölt állapot(ok)ba. A finomításnál említett módon, ha ez OR finomítás, akkor egy aktív állapot lesz, ha AND típusú, akkor minden konkurens régióban lesz egy. A példában kezdés után az s2 lesz aktív (a legfelső szintű kezdőállapot ide vezet), illetve hozzá tartozóan s21 és s22. Az egy pillanatban aktív állapotok halmazát nevezik az állapottérkép *konfigurációjának*.

Ezután következik az állapotgép egy lépése, melynek első fázisa, hogy az állapottérképhez rendelt eseménykezelő kiválaszt egy eseményt azok közül, amit az állapottérképnek küldtek (a választás módját a szabvány nem specifikálja). Megvizsgáljuk hogy az aktív állapotokból kivezető átmenetek közül melyeknek ez a trigger eseménye, majd ezek közül kiválasztva azokat, amelyeknek teljesül az őrfeltétele, megkapjuk az engedélyezett átmenetek halmazát. A példában, ha az e5 esemény érkezik, akkor az s2-ből s1-be vezető (legyen a neve t1), az s22-ből s23-ba (t2) és az s21-ből s25-be vezető (t3) lesz engedélyezett.

Ezek között lehetnek olyanok, melyek konfliktusban vannak, ugyanazt az állapotot akarják elhagyni (például t1 és t2, mindkettő elhagyja s22-t), ezek nem tüzelhetnek egyszerre. Lehetnek olyanok, amelyek tüzelhetnek egyszerre, például t2 és t3. Így a nem konfliktusban lévő, engedélyezett átmenetekből halmazok képezhetőek, amik potenciálisan szóba jönnek a tüzelendő átmenetek kiválasztásakor. Esetünkben {t1} és {t2, t3} a két halmaz. Ezeknek a halmazoknak tovább nem növelhetőeknek kell lenniük, azaz például {t2} nem lehetne önmagában egy halmaz.

A konfliktus feloldása elsőként prioritások segítségével történhet. Egy átmenetnek prioritása van egy másikkal szemben, ha a forrásállapota a hierarchia szintben mélyebben van, mint a másiké. Tehát t1 prioritása kisebb, mint t2-é, t2 és t3 prioritása azonos. Egy átmenet nem tüzelhet, ha van nála magasabb prioritású engedélyezett átmenet, ezt a szabályt figyelembe véve kapjuk a tüzelhető átmenetek halmazát, jelen esetben a {t2, t3} halmazt.

Ha több tüzelhető halmaz van, akkor azok közül véletlenszerűen választunk egyet, és eltüzeljük a benne lévő átmeneteket. Az átmenetek elhagyják forrásállapotaikat (a mélyebben fekvőtől kezdve a magasabb szinteken lévők felé haladva), ha ezekhez az állapotokhoz definiáltunk kilépési akciókat, akkor azokat végrehajtjuk, ezután következik maga az átmenet akciója, végül pedig belépünk a célállapotba, és elvégezzük az esetleges belépési akciókat, fentről lefelé haladó sorrendben.

A példánkban {t2, t3} tüzelne, de t1 tüzelése tanulságosabb, ezért vizsgáljuk most meg azt. Először s21 és s22 kilépési akciója, majd a1, végül s1 belépési akciója hajtódik végre.

Az UML állapottérképeken érvényes az úgynevezett *run-to-completion* elv, tehát a választott tüzelhető halmaz elemeinek tüzelésével még nem ért véget az esemény feldolgozása, ha van még olyan trigger nélküli átmenet a kialakult új konfigurációban, ami így engedélyezetté vált, akkor azokat is tüzelni kell, egészen addig, amíg nincs már több engedélyezett átmenet, és ezzel elértünk egy *stabil állapotot*.

Az állapottérképek tehát egy magas absztrakciós szintű, jól használható leírást adnak a rendszer megtervezéséhez.

Beágyazott rendszerek esetén további leíró formalizmusokat is alkalmaznak. Ilyen modellezési nyelv például az SCR (Software Cost Reduction), melyben táblázatos formában lehet megadni a rendszerrel szemben támasztott követelményeket. A módszer kidolgozója az amerikai Naval Research Laboratory rendelkezésre bocsátott egy szoftvercsomagot a modellezés segítésére, mely szimulátort, konzisztencia ellenőrzőt és verifikációs eszközt is tartalmaz. A diplomamunka későbbi fejezeteiben azonban az UML nyelvet fogjuk használni, így az SCR részletes leírására nem térek ki, azt pusztán csak megemlítettem, mint további példát beágyazott rendszerek esetén használt modellezési nyelvre.

1.2 Modell ellenőrző eszközök

A részletes modell elkészítése és manuális ellenőrzése azonban sok esetben nem elégséges. A bizonyítottan jól működő rendszerek iránti igény növekedésével megjelentek az informatikában a különböző formális módszerek [Pat03]. Az rendelkezésre álló egyre nagyobb számítási kapacitás pedig lehetővé tette, hogy a rendszer viselkedését leíró modelleket akár kimerítő módon „ellenőrizzék”, azaz teljes állapotterüket bejárják, minden állapotot megvizsgáljanak, és bizonyos fontos tulajdonságok meglétét vagy hiányát így bizonyítsanak. A ’80-as években jelentek meg először az ilyen módon működő ún. *modell ellenőrző* eszközök a hardvertervezésben, majd fokozatosan alkalmazták őket a protokollok és később a szoftverek ellenőrzésében is.

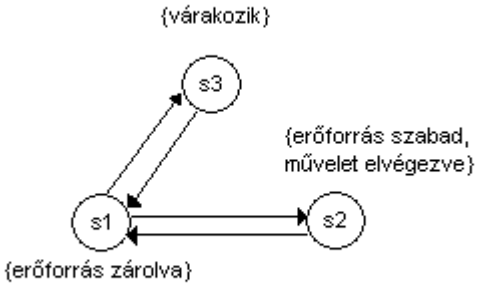
A modell ellenőrzés során a modell specifikálására valamilyen alacsony szintű, matematikailag jól kezelhető formalizmus szolgál, például a *Kripke-struktúra* vagy a *címkézett tranzíciós rendszer* (LTS – Labeled Transition System).

1.2.1 Felhasznált formalizmusok

A *Kripke-struktúra* M = (S, R, L) hármas, ahol

- S: állapotok véges halmaza,
- R: állapot-átmeneti reláció, $R \subseteq S \times S$,
- L: állapotok címkézése atomi kijelentésekkel.

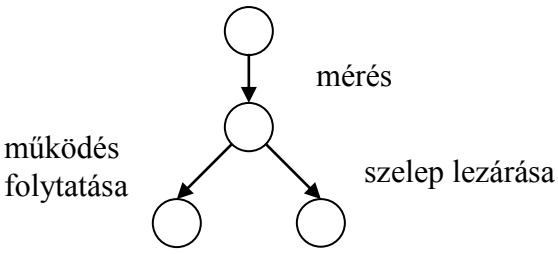
Az atomi kijelentések között olyan –a rendszerrel kapcsolatos– állítások szerepelnek, amelyek már tovább nem bonthatók. A 2. ábrán egy példát láthatunk a Kripke-struktúrára, mégpedig egy zárolással védett erőforrást használó rendszer modelljét. A rendszernek három állapota van (s1, s2 és s3), melyeket a „várakozik”, „erőforrás zárolva”, „erőforrás szabad” és „művelet elvégezve” atomi kijelentések valamilyen kombinációival címkézzük.



2. ábra: Kripke-struktúra

Az *LTS* pedig T = (S, Act, →) hármas, ahol
- S: állapotok véges halmaza,
- Act: akciók véges halmaza,
- →: címkézett állapot-átmeneti relációk, → ⊆ S x Act x S

A 3. ábrán egy egyszerű címkézett tranzíciós rendszert láthatunk.



3. ábra: LTS

A mérnöki modellezési nyelvekből a matematikai formalizmusok általában *modell transzformációval* származtathatók, pl. UML állapottérképből Kripke struktúra stb. (ld. lentebb). A gyakorlatban is használatos eszközöknek valamilyen saját, magasabb szintű leíró nyelvük van, de belső reprezentációként gyakran használják valamelyik fenti formalizmust, például a SPIN az LTS-t.

1.2.2 Temporális logikák

Az ellenőrizni kívánt tulajdonságok megadására az úgynevezett temporális logikák használatosak. Ezek az időbeliség, sorrendiség kezelésére *temporális operátorokat* vezetnek be (például „mindig igaz” vagy „valamikor igaz lesz”). A sokféle temporális logika közül a modell ellenőrzésben kettő használata terjedt el, ezek a PLTL és a CTL. Mindkettő kijelentés-logika, azaz atomi kijelentéseket, valamint az ÉS, VAGY, továbbá a NEGÁLÁS Boole logikai operátorokat használják a temporális operátorok mellett.

A *PLTL*-ben (Propositional Linear Time Logic) az idő lineáris. Azaz az idővonal egy olyan egyenes (a rendszer egymás utáni állapotainak sorozata), amin a temporális operátorok értelmezhetőek, s ezen egyenes adott pontjaiban vizsgáljuk az atomi kijelentések értékét.

A fontosabb operátorok:
- G p: p kijelentés minden időpillanatban igaz (General),
- F p: p kijelentés majd valamikor igaz lesz (Future),
- X p: p kijelentés a következő időpillanatban igaz lesz (a PLTL diszkrét időt feltételez, így értelmezett a következő időpillanat fogalma) (neXt),
- p U q: p minden állapotban igaz, amíg q igaz nem lesz (Until).

Nézzünk néhány példát LTL kifejezésekre és ezek intuitív jelentésére:
- G ($p \Rightarrow$ F q): minden állapotban igaz, hogy ha p igaz lesz, akkor utána valamikor q is igaz lesz (például, ha kiadunk egy kérést, arra mindig kapunk választ).
- F ($p \wedge$ X q): majd valamikor teljesül, hogy p igaz lesz és közvetlenül utána q is igaz lesz.
- FG p: valamikor olyan állapotba kerülünk, ami után p mindig igaz lesz (például a kezdeti átmenetek után beáll a rendszer egy stabil állapotba).

A *CTL* (Computational Tree Logic) a PLTL-lel szemben egy elágazó idejű logika, itt egyetlen időpillanathoz több rákövetkező tartozik, az idő (vagyis a rendszer által egymás után bejárható állapotok sora) a jelenből kiinduló fát alkot. A különböző ágak kezelésére két útvonal kvantor használatos:
- A: minden, az adott állapotból kiinduló úton,
- E: valamelyik, az adott állapotból kiinduló úton.

Az atomi kijelentések és az operátorok segítségével formulákat alkothatunk.
A CTL operátorok használatát a következő példák szemléltetik:
- EG p: létezik olyan lefutás (E), ahol a p kifejezés mindig teljesül (G) (ezt például nem tudjuk kifejezni LTL segítségével).
- AGEF p: bármelyik állapotban vagyunk is a rendszer futása során (AG), létezik olyan további végrehajtás (E), hogy elérünk majd valamikor (F) egy olyan állapotot, ahol p teljesül.

A PLTL és CTL pontos formális szintaktikája és szemantikája megtalálható [Pat03]-ban.

Tehát a *modell ellenőrzési probléma* a következő: egy M struktúrában a p temporális logikai kifejezés igaz-e. Ha az összes ilyen p-t keressük, akkor globális, ha csak egy adott konkrét állapotra vizsgáljuk a formula teljesülését, akkor lokális modell ellenőrzésről beszélünk.

A modell ellenőrzés legnagyobb kihívása, hogy egy viszonylag egyszerű modellnek is több százezer állapota lehet, és gyakorlati alkalmazáskor konkurens rendszerekben az egyes lokális állapotátmenetek nagyszámú lehetséges átlapolt végrehajtása miatt exponenciálisan növekszik az állapottér, így az állapottér robbanás jelensége következik be. Ezért nagy jelentőségű az állapottér hatékony tárolása, és a bejárás optimalizálása. Ezekről a technikákról a későbbi fejezetekben lesz szó.

1.2.3 Modell ellenőrzők használata mérnöki modellekre

A fejezet elején láttuk, hogy az állapottérképek nyelve nagy leíró erejű, jól használható módszert biztosít vezérlésorientált rendszerek modellezésére. A vizuális jelölőrendszer könnyen áttekinthetővé teszi a modellt, de –bár segít a helyes működés ellenőrzésében– azt igazolni nem tudja. Bizonyításra alkalmasak a modell ellenőrző eszközök, bonyolult tulajdonságok verifikálását is el tudják végezni. Az állapottérképeken megadott működést a modell ellenőrzőkben is le lehet írni, azonban az eszközök bemeneti nyelve általában kicsit eltér az állapottérképes leírásoktól, ad-hoc módon nehéz megadni egy összetett állapottérképet bennük. Itt jut komoly szerep a *modell transzformációs* módszereknek, amik *lehetővé* teszik két teljesen különböző modellezési nyelv között az automatikus konverziót, a két nyelv *metamodelljének* felhasználásával. Így lehetőségünk nyílik arra, hogy a mérnöki szemlélethez közel álló állapottérképes leírásban tervezzük meg a rendszerünket, ennek ellenére ne kelljen nélkülöznünk a formális módszerek által biztosított bizonyítási képességeket se. [BR04] áttekint és értékel kilenc, az irodalomban megjelent módszert, ami állapottérképek ellenőrzéséhez modell ellenőrzőt használ.

1.3 Beágyazott, eseményvezérelt rendszerek tesztelése

1.3.1 Klasszikus tesztelési fogalmak

A tesztelés metodikájának nagy múltú, átfogó irodalma van [Binder99], tekintsük át a legfontosabb fogalmakat. A tesztelés során a *tesztelendő rendszert* (SUT – System Under Test) bemeneti adatokkal látjuk el, megfigyeljük a kimeneteit, és ebből próbálunk arra következtetni, hogy a helyes, elvárt működést valósítja-e meg. A tesztelés célja hibák találása a rendszerben (nem pedig a helyes működés bizonyítása).

A rendszer ellenőrzését alapvetően két megfontolásból végezzük:

Verifikáció: azt szeretnénk ellenőrizni, hogy a rendszert helyesen tervezzük-e, azaz az adott tervezési lépés eredménye megfelel-e az előző fázisban elkészült specifikációnak.

Validáció: azt ellenőrizzük, hogy tényleg a kezdeti felhasználói igényeknek megfelelő rendszert valósítunk-e meg. A rendszerrel támasztott összes funkcionális és nem funkcionális (biztonság, teljesítmény, stb.) követelményt teljesít-e.

Tesztelést a fejlesztési életciklus több különböző fázisában alkalmazunk, a tervezési fázisban a specifikáció, a megvalósítási fázisban pedig az implementáció helyes és a követelményeknek megfelelő működését ellenőrizzük.

Az implementáció tesztelése során különböző szinteken végezhetjük az ellenőrzéseket. Először az elkészült önálló kis egységek (modulok, osztályok, komponensek, stb.) működését önmagukban, úgynevezett *egység (unit) tesztek* során teszteljük. A következő fázisban összekapcsoljuk a modulokat, ilyenkor az *integrációs tesztek* biztosítják, hogy a készülő, összetett rendszer is még helyes működést produkáljon. Végül az elkészült teljes rendszer *rendszertesztje* következik, melynek során elvégezzük a teljes rendszer verifikációját, majd a validációját. (Az itt felsoroltakon kívül még rengeteg speciális célú, általában valamilyen nem funkcionális követelményt ellenőrző tesztet is végrehajtanak, például terheléstesztet, stressz teszt, használhatósági teszt, stb.)

A tesztelésnek két alapvető megközelítése van, a funkcionális és a strukturális tesztelés.

Funkcionális tesztelés esetén a rendszernek csak a külső viselkedését ismerjük, az elvárt funkciók alapján tudunk teszteket készíteni hozzá, és a tesztek lefutását értékelni. A rendszer belső működéséről nem tudunk semmit, ezért szokás ezt a módszert fekete doboz (black-box) tesztelésnek is nevezni.
Strukturális tesztelés esetén ismerjük, és ki is használjuk a rendszer belső felépítését. Az egyes teszteket és a tesztelési célokat ennek megfelelően fogalmazzuk meg. Ennek a módszernek a másik elterjedt neve, utalva arra, hogy belelátunk a rendszerbe, fehér doboz (white-box) tesztelés.

A teszt fogalma túl általános, több elnevezést is takar, ezért pontosítsuk és differenciáljuk, hogy mit is értünk alatta a diplomamunka további részében:
Teszteset (test case): „bemeneti esemény – megfigyelhető, elvárt kimeneti esemény" lépéseket tartalmazó véges sorozat, mely a rendszer egy futását határozza meg.
Tesztkészlet (test suite): tesztesetek halmaza.
Fedési kritérium (coverage criterion): fontos, hogy egy tesztkészlet jóságát meg tudjuk határozni számszerűen, össze tudjuk hasonlítani egy másikkal. Ehhez nyújtanak segítséget a különböző fedettségi kritériumok. Ezek meghatározzák, hogy a tesztkészletnek bizonyos *tesztelési követelményeket* (test requirement) teljesíteni kell, például a tesztesetek lefuttatása során a forráskód adott sorát érinteni kell. Így tudunk egy mérőszámot rendelni a tesztkészlethez, hogy a fedettségi kritérium által meghatározott követelmények hány százalékát teljesíti (például, ha a forrássoroknak csak a 80 százalékát hajtatja végre, akkor az összes forrássor lefedése kritériumot 80%-ban fedi le a tesztkészlet). A fedettségi kritériumoknak két nagy csoportja van: a specifikációhoz és a programkódhoz rendelhetőek. További fedettségi kritériumokra látunk példát a második fejezet végén.

2 Modell alapú tesztelési módszerek

A klasszikus, implementáció alapú, javarészt informális módszereket használó tesztelés rendkívül erőforrás- és időigényes folyamat, sok benne a hibalehetőség. Még ha heurisztikákat és jól bevált ökölszabályokat alkalmazunk az összes lehetséges bemeneti sorozatból a lényegesek, a többi lefutást is reprezentálóak kiválasztására, akkor is tömérdek teszteset marad, amiket elő kell állítanunk. Az egyes tesztesetekhez tartozó részletes kimeneti sorozatok kézi meghatározása sok időt, és a készülő rendszer működésének alapos ismeretét kívánja.

Ezekre a problémákra nyújthatnak megoldást a specifikáció alapú tesztelési módszerek. Ilyenkor a rendszer valamilyen magas szintű, általában félformális modelljét hívjuk segítségül a tesztelés feladatainak automatizálásához, megkönnyítéséhez. Ilyen modell lehet például SCR nyelven megfogalmazott követelmény leírás, állapotgépes specifikáció, vagy a diplomaterv későbbi részében is használt UML állapottérképek.

A rendszermodell elkészítésének több haszna is van. A részletes modell felépítése közben kiderülhetnek a specifikációban rejtetten meglévő inkonzisztenciák, a kész modell segít abban, hogy a teljes fejlesztői gárda ugyanúgy értelmezze a követelményeket és a feladatot. A modellben egy helyen össze van gyűjtve a rendszer összes ki- és bemenet pontja, a kívülről elérhető interfészek és a rendszerrel kapcsolatos események, így könnyű áttekinteni, hogy mik azok, amit például egy funkcionális tesztelés során le kell fedni. A teljes tesztelési fázis tervezése és maguk a konkrét tesztesetek is a modellből indulnak ki, így az ellenőrzés folyamán azt vizsgáljuk, hogy az implementáció a specifikációban leírt funkcióknak megfelel-e. Ha a kódból indulunk ki (kódsor lefedettséget vizsgálunk, az elkészült modulokhoz, metódusokhoz készítünk teszteket), akkor egy helyes eredményt nyújtó teszt még nem feltétlenül garantálja, hogy az tényleg a specifikációnak megfelelő működést produkálja. Ezen kívül a formális specifikáció eléggé egzakt ahhoz, hogy gépileg feldolgozható legyen. Így a modell alapján automatizálhatók egyes fejlesztési és tesztelési lépések.

Az ilyen módszerek alkalmazása ugyan komolyabb szakértelmet kíván, mint az egyszerű, manuális tesztesetek leprogramozása, ugyanakkor, mivel a tesztelés általában a teljes költségvetés körülbelül felét teszi ki, a folyamat kis részének a kiváltása (automatizálása) is jelentős megtakarítást jelenthet.

2.1 Az irodalomban javasolt alkalmazások

A rendszer modelljének felhasználásával többféle módon lehet a tesztelést elősegíteni. Tekintsük át a fontosabbakat:

Teszt orákulum (oracle) előállítása: A modellt arra használjuk fel, hogy egy bemeneti sorozathoz megmondja, megjósolja (innen a módszer neve) a rendszer kimenetét [RAO92]. Például véletlenszerűen előállított bemeneti sorozatokhoz kiszámíttatjuk a kimenetet. Ilyen módszerrel kapott tesztkészlettel is viszonylag nagy kezdeti fedettséget lehet elérni. A módszer előnye, hogy kicsi az erőforrásigénye, és a lassabb módszereket már csak azokhoz a tesztelési követelményekhez kell bevetni, amiket ez az előzetes tesztelés nem fedett le.

Fedés meghatározása: Modell alapú tesztelési módszerekkel megoldható, hogy meghatározzuk egy már meglévő tesztkészlet fedését egy adott fedési kritériumhoz. Ammann és Black által javasolt [AB02] megoldásban a tesztesetből állapotgépet generálnak, ami leírja a rendszer működésének azt a részét, amit a teszteset meghatároz. Majd ezen egy modell ellenőrzővel a kritériumból meghatározott követelményeket ellenőrzik. A kielégített követelmények számát elosztva az összes követelmény számával pedig megkapjuk a teszt fedési mérőszámát.

Konformancia tesztelés: Azt vizsgáljuk, hogy az implementáció mennyire felel meg a specifikációnak. A modell és az implementáció megfelelő elemeit össze kell rendelni egymással, majd a kettőt párhuzamosan futtatva ellenőrizni kell, hogy nincs-e eltérés.

Tesztgenerálás: A tesztelés talán legnehezebb részét, a tesztgenerálást is lehet automatizálni. Ilyenkor általában vagy a tesztelő által megadott szempontok szerint, vagy valamilyen fedési kritériumnak megfelelően teljes teszteseteket generálunk, és ezekből kiválasztjuk azt az optimális méretű halmazt, ami megfelelő fedettséget nyújt. Minimális számú vagy hosszúságú tesztkészlet generálása NP-teljes probléma [HLSCU03], ezért legtöbbször megelégszünk csak valamilyen heurisztika alkalmazásával kapott, nem optimális tesztkészlettel.

2.2 Tesztgenerálási módszerek és eszközök

A diplomamunka további fejezetei a modell alapú tesztgenerálással foglalkoznak részletesen, ehhez először vizsgáljuk meg, hogy milyen elméleti eredmények születtek ezzel kapcsolatban, és milyen eszközök férhetőek hozzá. Következzen a témában íródott néhány fontosabb cikk ismertetése.

A Pennsylvania egyetem munkatársai Model-based Test Generation projektjükben [MTG] a következő módszert használják. STATEMATE állapottérképes (az UML-hez nagyon hasonló leírás, fő különbség, hogy az átmenet prioritáskezelése másképp működik) leírásokat transzformálnak modell ellenőrző nyelvére. Majd egy tesztelési követelményhez generálnak egy megfelelő temporális logikai formulát. A formula negáltját ellenőriztetik, és a kiadódó ellenpéldát tesztesetté konvertálják. Módszerüket a [HLSC01] cikkben fejtik ki, ez szolgáltatta az általam megvalósított tesztgenerátor program elvének alapját, melynek részleteit a harmadik fejezet fejti ki. Ugyanakkor a megvalósítás vizsgálatakor több probléma is felmerült a cikkben javasolt megoldással kapcsolatban. A transzformációhoz ők az állapottérképek szemantikáját Kripke-struktúrák segítségével adják meg, és ehhez definiálnak szabályokat, melyekkel SMV leírássá lehet konvertálni. A formalizálás azonban nem teljes, például a hierarchiaszintek között átvezető átmenetek hiányoznak, holott ezek kezelése korántsem triviális. Továbbá a cikkben szereplő példa SMV leírásban keveredik a konfliktus és a prioritás fogalma, és nincsen szó semmilyen eszközről, ami ezt a folyamatot automatizálná. Ezért használtam a diplomatervben bemutatott tesztgenerátor programomban egy másik [LMM99], bizonyítottan helyes formalizálást.

Az AGEDIS (Automated Generation and Execution of Test Suites for DIstributed Component-based Software) [AGEDIS] egy Európai Uniós három éves kutatási projekt, melynek célja a szoftverek minőségének javítása a tesztelés automatizálásával. Az akadémiai és üzleti partnerekből álló konzorcium kidolgozott egy tesztelési metodológiát a hozzá tartozó részletes specifikációkkal, valamint kifejlesztette a

szükséges eszközöket. Az általuk javasolt módszerben [Dav03] a tesztelő egy UML kiegészítésben (AML Profile) teszt direktívákat ad meg először (melyik állapotokat kell érinteni, melyik legyen a kezdő és a végállapot, stb.). Ezután a modellt egy köztes nyelvre (IF – Intermediate format) transzformálják, ez lesz a tesztgenerátor bemenete. Az általuk elkészített tesztgenerátor alapja az IBM nem nyilvános, Gotcha nevű [FHNS02] és a Verimag cég TGV nevű eszköze. A modell belső reprezentációjaként címkézett tranzíciós rendszert (LTS) használnak. Az így kapott absztrakt tesztkészletet az implementációhoz kapcsolják, és egy teszt futtatókörnyezetbe táplálják. Az esetleges hibás futásokat pedig egyrészt visszacsatolják a tesztgenerátorba, másrészt megjelenítik a kapott futást. A projekt dokumentációinak nagy része letölthető, és az elkészült eszközök is elérhetőek 2004. július végétől.

P. E. Black és munkatársai a National Institute of Standards and Technology Automatic Test Generation from Formal Specifications projektjében [ATG] a *mutációs analízis* technikát kombinálják modell ellenőrző használatával a tesztgeneráláshoz [ABM98]. Mutációs operátorokat definiáltak (pl. feltétel negálása, másik konstans használata), amik úgy változtatják meg a specifikációt, hogy az más kimenti eredményeket adjon. Ezekhez temporális logikai formulákat rendelnek, amik jellemzik ezt a mutációt. Majd a formulát ellenőrzik az eredeti modellen, és mivel a mutáns különbözik az eredetitől, valahol sérülni fog a mutánsra jellemző formula. A kapott ellenpéldákból generálják a teszteseteket, amik olyan futásokat reprezentálnak, amik képesek a mutációt okozó hibát felderíteni. A módszer demonstrálásához egy több eszközből álló környezetet készítettek, mely az SMV modell ellenőrzőt használja, és a TAO (Test Assistant for Objects) tesztelési keretrendszer bemeneti formátumára konvertálja a teszteseteket.

Az *absztrakt állapotgépeket*, mint modellezési nyelvet többen is használták automatikus tesztgenerálás során. A Microsoft Research Foundations of Software Engineering csoportja [AsmL] kifejlesztett egy modellező nyelvet (AsmL – Abstract State Machine Language), mely tulajdonképpen ASM-ek végrehajtható leírása. Továbbá készítettek hozzá egy eszközt (AsmL Test Tool, [GNTV03], [Gries03]), mely, többek között konformancia tesztelésre és tesztgenerálásra használható. Az AsmL nyelven elkészített modellt a tesztelő először előkészíti, definiálja, hogy az egyes függvények paraméterei milyen tartományból származzanak. Ezután megad tulajdonságokat, amelyek levágják az állapottér bizonyos részeit, ezzel elkerülve az állapottér robbanás problémáját. Ugyanis a tulajdonságok megadásával absztrakt állapotok keletkeznek, amelyekbe több valós állapot tartozik, a bejárás során egy ilyen elérése után bejártnak tekinti az összes többi ide tartozó valós állapotot is. Következő lépésként az absztrakt állapotgépből a program egy véges állapotgépet generál, és ezt elkezdi bejárni, amíg bizonyos állapotba nem kerül a rendszer vagy például adott modell fedettséget el nem ér. A bejárás során kapott futások lesznek a tesztesetek. Az AsmL nyelv a Microsoft .NET keretrendszerre épít, így lehetséges, hogy a modell implementációjaként megadható egy tetszőleges .NET-es program, és annak konformanciáját lehet vizsgálni, a generált véges állapotgép egy minimális költségű bejárásával.

Gargantini és társai [GRR03] cikkükben bemutatnak egy módszert tesztek generálására ASM modellekből SPIN modell ellenőrző segítségével. A módszer menete hasonló a korábban bemutatott [HLSC01]-hoz, itt is először definiáltak egy transzformációt. Majd ASM-eken értelmezett fedési kritériumokból temporális logikai formulákat generálnak és ezekhez ellenpéldát keresnek. Elkészítették a tesztgenerátor

program prototípusát [ATGT], és beszámolnak egy kis modellen végzett sikeres próbáról is.

Jeff Ofutt és Aynur Abdurazik tesztgenerálási módszere SCR követelmény-leírásokból indult ki, később ezt dolgozták át [OA99], hogy UML állapottérképek bizonyos fajtájára is alkalmazható legyen. Olyan állapottérképekre szorítkozik, ahol az átmeneteket úgynevezett „change event”-ek triggerelik (ez olyan esemény, ami akkor váltódik ki, ha a hozzá megadott Boole logikai feltétel megváltozik). Az általuk elkészített TCGen programmal egy, az irodalomban gyakran használt példán (Cruise Control) végeztek méréseket, és átlagosan 56%-át találta meg az injektált hibáknak a generált tesztkészlet.

Az eddigiektől kicsit eltérő megközelítést követtek B. Legeard és társai [Leg02]. Az ő BZ-Testing Tool eszközük B vagy Z nyelven megadott modellekhez generál teszteket a boundary-test elv segítségével. (A módszer lényege, hogy olyan bemeneteket generálunk, amik a bemeneti tartomány széléről veszik fel az értéküket, és olyan állapotba visszük a rendszert, ahol annak valamelyik belső változója az értelmezési tartományának határán van.) Ehhez a háttérben nem modell ellenőrzőt használ, hanem a specifikációt egy kényszer-kielégítési problémává alakítja, és az általuk fejlesztett CLPS-BZ Solver eszközzel megoldja azokat. A programot ipari alkalmazásokban is tesztelték (pl. Smart Card szoftver), és LEIRIOS Test Generator néven kereskedelmi változata is létezik.

A fentebb bemutatott eszközökön kívül természetesen léteznek még tesztgeneráló programok. Ezeket tekinti át [Hart03], részletezve az egyes szoftverek elérhetőségét, a használt formalizmusokat és be- és kimeneti formátumokat.

Az irodalomban szereplő további, modell alapú tesztelési módszereket használó esettanulmányokat foglal össze az 1. táblázat.

Az első oszlop mutatja, hogy nagyon széles az alkalmazások skálája, ahol a különböző módszerek alkalmazhatóak. Az egyes esettanulmányok jelentősen eltérnek egymástól mind az alkalmazott modellezési formalizmusban, mind a feladat méretében. A mintapéldák igazolják, hogy nagy rendszerek esetén is lehet létjogosultsága a modell alapú megközelítésnek, a modell méretéből adódó korlátokat át lehet hidalni.

1. táblázat: Modell alapú tesztelésről szóló esettanulmányok az irodalomban. Forrás: DACS Gold Practice, http://www.goldpractices.com/practices/mbt

Alkalmazási terület	*Leírás*	*Metrikák*
Power PC	Szakértő rendszer használata, mely a processzor architektúra modelljét és heurisztikákat tartalmaz.	1530 hibát találtak 3 processzorban
Óra alkalmazás Pocket PC-re	12, 49 és 864 állapottal rendelkező modellen adat alapú kritériumok ellenőrzése.	10 szándékos hiba megtalálása
Mars Polar Lander	50 sornyi programhoz tesztek generálása. A tesztek megtalálták a hibát, mely a misszió kudarcát okozta.	19 teszt, 12 órányi modellezési munkával
Biztonság	Tesztek generálása SCR nyelven megfogalmazott biztonsági követelményekhez. A teszteket később Oracle és InterBase adatbázisokon futtatták.	40 teszt vektor
Távközlés	Állapot alapú modell digitális rendezőhöz.	80%-os hatékonyság növekedés
Távközlés	Tesztek generálása a Bellcore cég Intelligent Services Control Point eszközéhez, és az általa kezelt üzenetformátumokhoz.	6100 teszt, 440 hiba
Távközlés	Négy esemény tanulmány több millió sornyi kóddal, tesztek generálása statisztikai eszközökkel.	-
Pocket PC alkalmazás	Tesztek generálása öt komponenshez véges állapotgépek segítségével.	-
POSIX és Java standardok	POSIX szabvány egy részét és a Java kivételkezelését véges automatákkal modellezték.	9 teszt a POSIX-hoz, 6914 Java kód
IP telefonok, Call Center API	Öt mintakísérlet az IBM-nél. Véges állapottérképeket használtak a tesztekhez.	-
Cruise Control rendszer	400 sornyi C kódhoz 54 teszt generálása, fedési kritériumok alapján	24 injektált hiba megtalálása

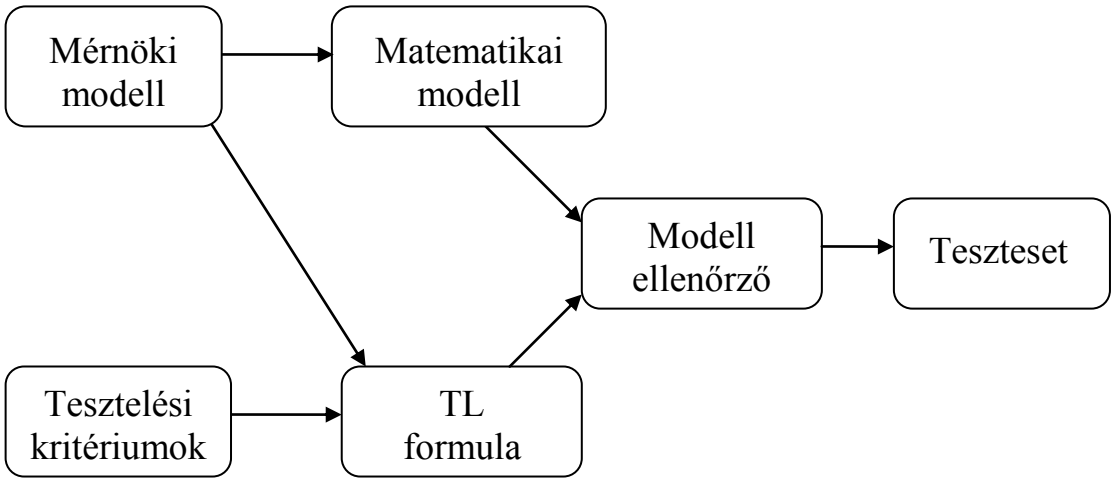
Láthatjuk tehát, hogy az automatikus tesztgenerálás területén aktív kutatás folyik, sok a témával kapcsolatos friss cikk és tudományos eredmény. Jelentek meg már nyilvános eszközök a támogatására, de ezek nagy része még prototípus, és a módszerek megvalósíthatóságával kapcsolatban még nagyon sok a nyitott kérdés. A diplomaterv további fejezetei ezek egy részére keresnek választ egy tesztgenerátor program implementációjával és annak vizsgálatával.

2.3 Automatikus tesztgenerálás modell ellenőrzővel

Az ismertetett módszerek közül az itt következőt választottam alapul a tesztgeneráló alkalmazás megvalósítása során.

2.3.1 Az automatikus tesztgenerálás módszere

A módszer főbb lépéseit a következő ábra szemlélteti:



4. ábra: Automatikus tesztgenerálás

- Egy kiválasztott fedési kritérium alapján az egyes tesztelési követelményekhez temporális logikai formulákat rendel. A kritériumoknak megfelelő formulákat a fejezet későbbi része ismerteti.
- A rendszer állapottérképekkel adott modelljét modell ellenőrző bemeneti nyelvére transzformálja. A transzformációt a harmadik fejezet részletezi.
- Olyan futásokat szeretnénk kapni, amikben az egyes tesztelési követelmények teljesülnek, így a kapott formulák negáltját ellenőrizteti egyesével. A tesztgenerálás igényeinek megfelelő ellenőrzési beállításokkal az ötödik fejezet foglalkozik.
- Végül a modell ellenőrző futásainak kimenetét (ellenpélda) tesztsorozattá konvertálja, a lényeges elemeket kiválogatja belőle. Ennek a menete a negyedik fejezetben található.

2.3.2 Fedettségi kritériumok megadása LTL formulákkal

Tekintsünk át néhány gyakran használt, közvetlenül UML állapottérképeken értelmezhető tesztelési kritériumot és ezek LTL megfelelőjét (a megadott formulák könnyedén átírhatóak CTL formulákká, legtöbbször csak az E útvonal kvantort kell beírni az F operátor elé).

2.3.2.1 Vezérlés alapú kritériumok

A kritériumok megadásánál a következő jelöléseket használjuk:

- ! : negálás,
- $\wedge$: ÉS logikai operátor,
- $\vee$: VAGY logikai operátor,
- F, U, X : temporális operátorok,
- S : az állapottérkép állapotainak halmaza,
- T : az állapottérkép átmeneteinek halmaza.

Teljes állapotfedés: Az állapottérkép minden egyes állapotához szeretnénk egy olyan tesztsorozatot kapni, ami elvisz abba az állapotba. Ehhez a következő formulahalmazt kell ellenőriztetni:

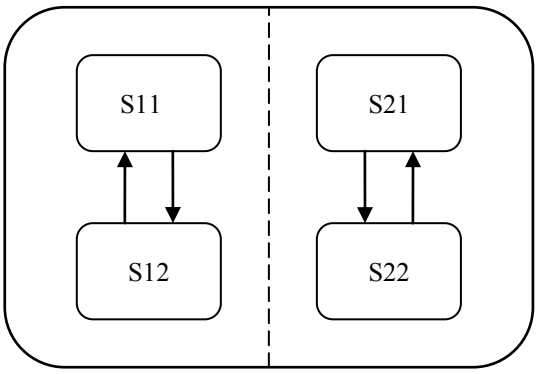
$$\{ \; ! \; (F \; in(s)) \; | \; \forall \; s \in S \; \}$$

ahol in(s) akkor igaz, ha az s állapotban vagyunk, egyébként hamis (ez az állapotváltozókon értelmezhető egyszerű Boole logikai kifejezés rövidítéseként fogható fel).
Az F in(s) formula ugyanis akkor igaz a modellen, ha valamikor el tudunk jutni az s állapotba. Ennek a negáltját ellenőriztetve pont egy olyan ellenpéldát kapunk, ami megmutat egy ilyen utat (ha létezik egyáltalán).
Tehát például az 1. ábrán lévő állapottérkép esetén a teljes állapotfedés a következő formulák ellenőrzését jelenti:

```
!(F in(s1)), !(F in(s2)), !(F in(s21)), !(F in(s22)), !(F
in(s23)), !(F in(s25))
```

Ezt a hat darab formulát egyenként ellenőriztetve az állapottérképnek megfelelő modellen hat darab ellenpéldát kapunk, amelyek megfelelnek az egyes állapotokat tesztelő teszteseteknek.

Teljes konfigurációfedés: Ha az összes lehetséges konfigurációt bejárjuk, akkor egy bővebb működést ellenőrizhetünk, mint a teljes állapotfedéskor. Hisz vegyünk példaként egy, az 5. ábrán szereplő konkurens régiót, ami kettő-kettő állapotot tartalmaz, ezek egymástól függetlenül válthatnak. Két teszt is képes teljes állapotfedést produkálni (az egyik {S11, S22} állapotba viszi a rendszert, a másik {S12, S21} állapotba). Viszont teljes konfigurációfedéshez négy különböző teszt kell ({S11, S21}, {S11, S22}, {S12, S21} és {S12, S22}).



5. ábra: Konkurens régió állapottérképen

A temporális logikai formula a következő:

$$\{!\ F\ in(c)\ |\ \forall\ c \in C\ \}$$

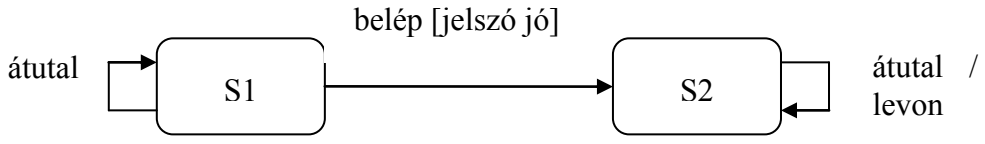
ahol C a lehetséges konfigurációk halmaza, in(c) pedig egy konfigurációban tartózkodást jelöli, azaz $in(c) = in(s_1) \wedge in(s_2) \ldots \wedge in(s_n)$.

Teljes állapotfedés: Szeretnénk az állapottér minden egyes átmenetét legalább egyszer végrehajtani:

$$\{\ !\ (F\ firing(t))\ |\ \forall\ t \in T\ \ \}$$

ahol firing(t) akkor igaz, ha a t átmenet tüzel, egyébként hamis.

Implicit átmenetek fedése: Ha azt is szeretnénk tesztelni, hogy a rendszer véletlenül se reagál olyan eseményekre, amiket nem vettünk fel az állapottérképre, akkor teljessé kell tennünk azt. Azaz önhurkokat kell felvenni az egyes állapotokhoz minden egyes, eddig ott nem szereplő (átmenetet nem triggerelő) eseménnyel, és utána az átmenetfedésnél látott formulákat kell ellenőrizni. Ez sok esemény esetén nagyon erőforrás-igényes lehet, viszont bizonyos speciális eseményekre érdemes lehet külön elvégezni ilyen tesztek generáltatását. Például, a 6. ábrán szereplő állapottérképen érdemes lehet vizsgálni, hogy átutalást csak akkor tudjanak végezni, ha beléptek a rendszerbe. Azaz felveszünk egy önhurkot S1-be, amit az átutal esemény triggerel, erre keresünk egy tesztet, és a tesztelés során ennek futtatásakor az az elvárt eredmény, hogy ne hajtódjon végre a művelet.



6. ábra: Implicit átmenet felvétele

Átmenet pár fedés (transition pair coverage): Egy állapot minden be- és kimenő átmenet párjának egymás utáni tüzelését megvizsgáljuk:

$$\{!(F(firing(t_i) \ \&\ X(firing(t_j)))) \mid \forall s \in S, \forall t_i \in s_{in}, \ \forall t_j \in s_{out}\}$$

ahol s_{in} az s állapotba bevezető, s_{out} az abból kivezető átmenetek halmaza.

További vezérlés alapú fedési kritériumok találhatók [GRR03]-ban (pl. All guard condition – minden őrfeltétel miden egyes tagját vizsgáljuk igaz és hamis értékekre) és [HLCS01]-ben.

2.3.2.2 Adat alapú kritériumok

Az UML állapottérképeken szerepelhetnek változók is, ezeket figyelhetjük az őrfeltételekben, vagy értéküket megváltoztathatjuk az akciókban. Alapos tesztelés során nem elég pusztán a vezérlési részeket végigjárnunk, fontos, hogy az adatok helyes kezelését is ellenőrizzük.
Ehhez definiáljuk a következő fogalmakat:

- **def(v)**: Azon átmenetek halmaza, ahol az átmenet akciójában a *v* változónak értéket adunk.
- **use(v)**: Azon átmenetek halmaza, ahol az átmenet őrfeltételében vagy akciójában a *v* értékét felhasználjuk.

Minden értékadás (All-def coverage): az értékadás helyességét ellenőrizhetjük egy olyan futással, ahol az értékadás után megvizsgáljuk az első használat során kapott választ:

$$\{!F(\ firing(t)\ \&\ X\ (!\ d(v)\ U\ (u(v)))\ )\ \mid\ \forall\ v,\ \forall\ t\ \in\ def(v)\}$$

ahol $d(v) = \vee_{t \in def(v)}$ firing(t) és $u(v) = \vee_{t \in use(v)}$ firing(t).
Azaz nem igaz az, hogy valamikor tüzel a t átmenet, és utána nem tüzel olyan átmenet, ami v-nek értéket adna egészen addig, amíg fel nem használja valaki v-t. Az Until-os konstrukció használatára tehát azért van szükség, hogy biztosan t hatását ellenőrizzük.

Minden használat (all-use coverage): minden egyes változó-használathoz külön tesztet generálunk:

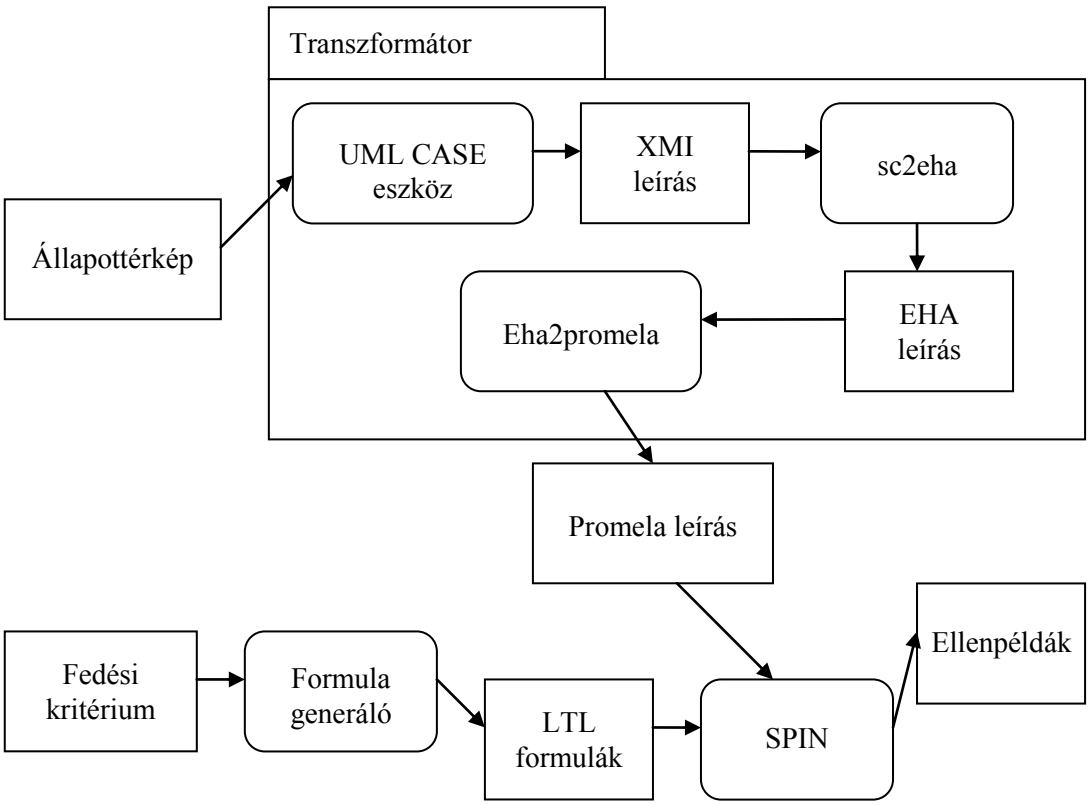
$$\{!F(firing(t)\&\ X\ (!\ d(v)\ U\ t'))) \mid \forall v, t \in def(v), \forall t' \in use(v)\}$$

Azaz minden változó minden egyes használatához készítünk egy formulát. A formula azt fejezi ki, hogy nem igaz az, hogy valamikor a jövőben tüzel egy, a változónak értéket adó átmenet (t), majd tüzel a tesztelendő t' átmenet. Közben pedig nem történik értékadás az adott változónál.

A [HLSCU03] cikkben találhatók további adat alapú fedési kritériumok.

3 Tesztgenerálási keretrendszer

Az előző fejezetben láttuk a tesztgenerálás általános folyamatát. Az elkészült program két külső eszközt használ, az állapottérkép → Promela transzformátort, és a SPIN modell ellenőrzőt. Mindkettő bonyolult funkcionalitást valósít meg, és több éve fejlesztik őket, ezért döntöttem úgy, hogy felhasználom őket. Működésük megértése fontos a tesztgenerálás lépéseinek bemutatásához, így álljon itt egy rövid ismertető.



7. ábra: Tesztgenerálási keretrendszer

3.1 A SPIN modell ellenőrző

A SPIN modell ellenőrző [Holz97] fejlesztését az 1980-as években kezdték a Bell Laboratories-ban, 1991-ben szabadon elérhetővé tették, és a mai napig is folyamatosan fejlesztik. Az eszköz célja szoftverek verifikálása, azon belül pedig főleg elosztott rendszerek és protokollok ellenőrzésére használják. Élőségi (pl. holtpontmentesség) és biztonsági tulajdonságok teljesülését is lehet vele vizsgálni, a kritériumok megadhatóak LTL kifejezések formájában is. On-the-fly módon dolgozik, azaz nem kell a teljes állapotteret felépítenie és bejárnia, mielőtt hozzákezd a tulajdonságok ellenőrzéséhez. Ez a tesztgenerálás szempontjából nagyon hasznos, hisz mi általában rövid ellenpéldákat keresünk, amihez nincs szükség a teljes állapottér hosszú ideig tartó felépítésére.

3.1.1 Automata alapú modell ellenőrzés

A modell ellenőrzési probléma megoldásának egyik a gyakorlatban használt módja az *automataelméleti megközelítés* [VW86], [Pat03]. Ez a módszer azon alapul, hogy minden Kripke-struktúrához konstruálható egy olyan A_M automata, mely pontosan azokat a szavakat fogadja el, amik megfelelnek a Kripke-struktúra egyes futásainak. Az automata ábécéje 2^{AP}, ahol AP a Kripke-struktúra atomi kijelentései. Hasonló módon minden PLTL formulához konstruálható egy A_P automata, ami pontosan azokat a szavakat fogadja el, ahol a formula teljesül. Így a modell ellenőrzési probléma ekvivalens

$$L(A_M) \subseteq L(A_P)$$

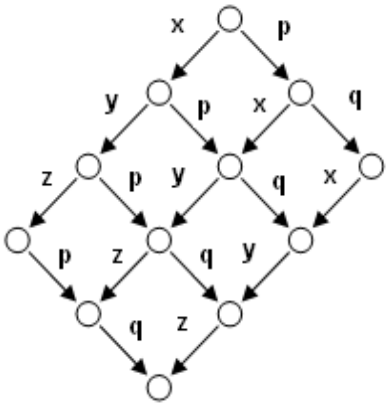
tartalmazási problémával, ahol L(A) jelöli az A automata által elfogadott nyelvet. Könnyebb ellenőrizni, ha a fenti kifejezést átírjuk a következő formára:

$$L(A_M) \cap L(A_P)^c = 0$$

Ezt pedig úgy lehet vizsgálni, hogy megnézzük, hogy az A_M és A_P^c automaták szorzat-automatája az üres nyelvet fogadja-e el, azaz a kiinduló állapotaiból elérhető-e elfogadó állapota.

A PLTL formuláknak megfelelő automatáknak a végtelen hosszú szavak elfogadását is el kellene tudni dönteniük (gondoljunk csak egy G(p) alakú formulára), ezért a klasszikus véges automaták helyett *Büchi automatákat* használnak. A Büchi automaták elfogadási feltételeit úgy definiálták, hogy képes legyen a végtelen hosszú szavakat is kezelni. A SPIN nyelvében az úgynevezett never claim konstrukció valósítja meg a PLTL kifejezésekhez tartozó Büchi automaták működését, egy PLTL formula megadásakor az eszköz automatikusan egy ilyet generál (never claim-re láthatunk példát a függelékben).

```
A:              B:
  x = 1;          p = 3;
  y = 2;          q = p + 1;
  z = x + y;
```



8. ábra: Részleges rendezés

Az automataelméleti módszer használatakor a legfontosabb optimalizációs technika az úgynevezett *részleges rendezés* (partial order reduction). Ennek alapgondolata, hogy párhuzamos rendszerek esetén az egymással átlapoltan végrehajtott műveletek minden lehetséges sorrendben végrehajtott megvalósítása helyett elég csak bizonyos reprezentatív végrehajtásokat megvizsgálni, azokat, amelyeknek kívülről is látható, hogy különböző eredménye van. Például a 8. ábrán látható két párhuzamosan futó

folyamat összes lehetséges tíz különböző lefutása helyett, mivel nem befolyásolják egymást a műveleteik, elég csak egyet figyelembe venni. A minimális méretű reprezentatív végrehajtás meghatározása túl nagy számításigényű lenne, de megfelelő heurisztikák alkalmazásával is már jelentős csökkenés érhető el az állapottér méretében és a verifikáció hosszában.

A SPIN modell ellenőrzőnek még rengeteg optimalizációs lehetősége van, memóriatömörítés, rövid futások keresése, és az úgynevezett *bitstate hashing* mód, amikor közelítő eredményeket kapunk, mindezekről az ötödik fejezetben lesz szó részletesen.

3.1.2 A verifikáció menete

Egy teljesen manuálisan, parancssori felület segítségével elvégzett verifikáció a következő lépésekből áll:

- Modell elkészítése Promela nyelven tetszőleges szövegszerkesztőben. Az elkészült leíráson véletlenszerű vagy interaktív szimulációkat futtathatunk, ezzel ellenőrizve, hogy helyesen készítettük-e el a modellt.
- PLTL formulából never claim generálása, pl.

  ```
  spin –f formula > formula_file_nev
  ```

  ahol a formula fájlnak tartalmaznia kell már a formulában használt atomi kijelentések megfeleltetését Promela-beli Boolean értékű kifejezésekre, a C makrókból ismert #define kezdetű sorok segítségével.
- A tényleges verifikációt egy a specifikációból és a formulából közösen generált program végzi, ennek forrását generálja a

  ```
  spin –a –N formula_file_nev modelle_file_neve
  ```

  A generált forrás neve pan (ami a *protocol analyzer* kifejezésből származik).
- Az elkészült C forrásfájlok fordítása valamilyen fordítóval. E lépés során lehet a verifikációs beállítások egy részét módosítani, például, hogy használjon-e valamilyen memóriatömörítési eljárást a modell ellenőrző.
- A kapott pan bináris futtatása. Ennek során a modell ellenőrző (maga a pan program) a generált állapotteret mélységi vagy szélességi bejárással bejárja, és ellenőrzi, hogy sérülnek-e a megadott feltételek. A bejárás tulajdonképpen a PLTL követelménynek megfelelő automata (never claim) és a modell automata „szorzatának” generálásával történik. Ha talál a követelménynek nem megfelelő futást, akkor azt ellenpéldaként megjeleníti. Ezen kívül az ellenőrzés általános információit is megjeleníti a végén, például mennyi memóriát használt, volt-e nem érintett kódrészlet, stb. A pan indításakor is lehetőségünk van további paraméterek beállítására, például a bejárás mélységének korlátozására.

3.1.3 A Promela nyelv

A SPIN a modellek leírására a Promela (PRocess MEta Language) nyelvet használja [Spin]. A nyelv kommunikáló, konkurens folyamatok leírására szolgál, így alapszerkezete a *processz*. A szokásos változótípusok és vezérlési szerkezeteken kívül a nyelv eleme még a *csatorna*, mely a globális változók mellett a processzek közötti kommunikáció eszköze. A csatorna méretét nullára állítva szinkron működésűvé válik az alapértelmezésként aszinkron csatorna.

Nézzünk egy egyszerű példát, melyben egy termelo nevű processz sorban az egész számokat küldi egy fogyaszto nevű processznek, ami csak annyit tesz, hogy kiolvassa a csatorna tartalmát, és beállít egy globális változót az érkezett számnak megfelelően.

```
chan csatorna = [2] of {int});
bool paros = false;

active proctype termelo()
{
      int i = 0;

      do
      ::    csatorna!i;
            i++;
      od;
}

active proctype fogyaszto()
{
      int uzenet = 0;

      do
      ::    csatorna?uzenet;
            if
            ::    (uzenet % 2 == 0) -> paros = true;
            ::    else -> paros = false;
            fi;
      od;
}
```

Egy Promela program működése a következő: Megvizsgáljuk, hogy a futó processzeknek (az active kulcsszó szerepel a deklarációjukban vagy valaki a *run* utasítással elindította őket) mi a soron következő utasításuk. Ha ez nem végrehajtható (például, olyan csatornából akarunk olvasni, amiben nincs üzenet, olyan feltétel szerepel benne, ami nem igaz), akkor az a processz blokkolódik. Ha egy processz aktuális utasítása egy *if* szerkezet, akkor az összes ágat megvizsgálja, és aminek nincs az első utasítása blokkolva, azok mind belekerülnek a végrehajtható utasítások halmazába. A nem blokkolt processzek utasításai közül választ egyet a futtató, és végrehajtja annak az utasítását. A *do od* szerkezet is hasonlóan működik, azaz véletlenszerűen választ egyet a nem blokkolt ágai közül, csak ez a végrehajtása után újra és újra végrehajt egyet, amíg valahol a *break* kulcsszóval ki nem lépünk belőle. Ezek a választások biztosítják a nem-determinisztikus működést.

A konkurens működéshez kapcsolódóan két speciális konstrukciója van még a Promela nyelvnek, az atomic és a d_step kulcsszó. Mindkettő atomi működést valósít meg, azaz a benne szereplő utasításokat az adott processz egyszerre hajtja végre, azokat nem szakíthatja meg másik processz. Ennek a segítségével lehet például kölcsönös kizárást megvalósító szerkezeteket létrehozni. A d_step ezen felül lépések oszthatatlan, determinisztikus sorozatát jelenti, amik a verifikációkor létrejövő automata egy átmenetébe képződnek le, ezért helyes használatával a verifikáció jelentősen gyorsítható.

3.2 Az SMV modell ellenőrző

Az elkészült tesztgenerátor programban ugyan a SPIN modell ellenőrzőt használtam, azonban korábban az SMV használatával is végeztem tesztgenerálást, és az ötödik fejezetben lévő méréseknél utalok rá, így célszerű itt röviden áttekinteni ennek az eszköznek a működését is. Az SMV program a gyakorlatban használt másik módszer, a szemantikán alapuló modell ellenőrzést [McM93] implementálja. A módszer lényege, hogy meghatározza, hogy az adott p kifejezés mely állapotokban igaz (jelöljük ezt a halmazt Sat(p)-vel), majd ellenőrzi, hogy a kérdéses állapot eleme-e Sat(p)-nek. A Sat(p) halmaz kiszámításában iteratív módszert alkalmaznak, melyben a CTL operátorok szemantikáját használják az egymást követő lépésekben a végleges Sat(p) halmaz meghatározásához.

A fentebbi módszer hátránya, hogy a köztes iterációs lépések során nagyméretű halmazokat kell tárolni, és rajtuk halmazműveleteket végezni. Ezért a halmazok helyett azok úgynevezett karakterisztikus függvényével dolgozik, Boole-függvényeket feleltet meg az egyes halmazoknak. Ezeknek az ábrázolására pedig létezik egy nagyon tömör módszer, a bináris döntési diagramok (BDD), ezek segítségével a szimbolikus modell ellenőrzési technikák is igen hatékonyak.

Az SMV modell ellenőrző nyelve alapvetően állapotgépek megadására szolgáló konstrukciókat tartalmaz:

```
MODULE Sender(s2r_in, r2s_out)
VAR
  state : { get, send, wait_for_ack };
  data : 0..15;--DATA
ASSIGN
  init(state) := get;

  next(state) :=
    case
      state = get                  : send;

      state = send & (s2r_in.tag = mt)   : wait_for_ack;
  ...
SPEC  AG AF (sender.state = get)
SPEC AF (receiver.state = deliver)
```

A MODULE kulcsszó segítségével lehet egy automatát definiálni, melynek változóit a VAR kulcsszó után kell felsorolni. Az ASSIGN részben következik a változók értékadása. A változók kezdeti értékére az init(változó), az aktuális értékére a változó kifejezéssel lehet hivatkozni, míg az automata lépése utáni következő értéket a *next(változó)* kifejezésnek értéket adva lehet beállítani. A SPEC részben pedig az ellenőrizendő CTL kifejezéseket kell felsorolni.

Az SMV is karakteres felületet biztosít az interaktív szimulációkhoz és verifikációkhoz. CTL kifejezések ellenőrzése esetén a modell beolvasása után kisimítja a modellt és felépíti a szükséges BDD-ket, majd elvégzi a verifikációt. Ehhez a művelethez nem készít külön programot, mint a SPIN. Másik, a tesztgenerálásnál hasznos tulajdonsága, hogy képes több formulát is ellenőrizni egy futás során az adatstruktúra újrafelhasználásával. A kiadódó ellenpéldák formátuma az újabb verziókban már lehet akár XML is.

3.3 Állapottérkép → Promela transzformátor

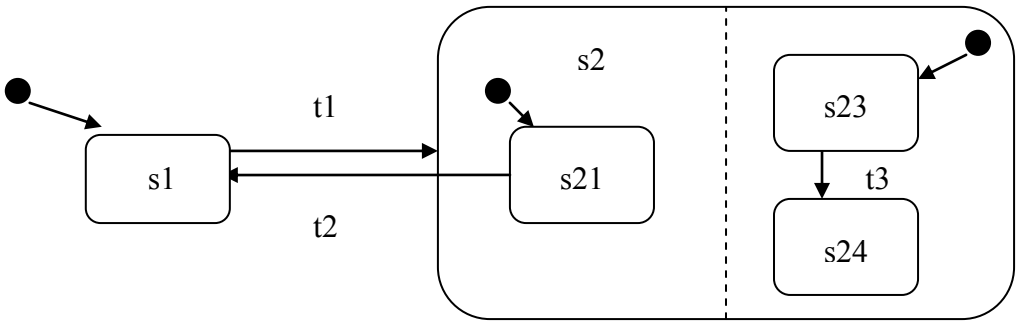
Az állapottérkép transzformációját a [LMM99] cikkben leírt módszer alapján két, a tanszéken készült program végzi. A transzformációt az nehezíti, hogy bár az UML nyelvhez megjelent szabvány, de ez csak az állapottérképek szintaktikáját rögzíti, a szemantikát pusztán természetes nyelven (angol mondatok formájában) adták meg, szabványos formális leírása nincsen. A kényelmes, magas absztrakciós szintű konstrukciók (pl. hierarchia szintek közötti átmenet, prioritási rendszer) átalakítása nem triviális feladat. Ezért a cikk a következő módszert javasolja: Transzformáljuk először az UML állapottérképet egy köztes, alacsonyabb szintű, de kellőképp formalizált leírásra (ez az úgynevezett *kiterjesztett hierarchikus automata*, EHA modell), és ezen már könnyebb megfogalmazni a transzformáció szabályait és bizonyítani a helyességét.

A *kiterjesztett hierarchikus automata* a klasszikus állapotgép bővítése a hierarchikus finomítás lehetőségével. Egy gyökér automatából és további automatákból áll, melyek a gyökérautomata vagy egymás állapotait finomítják tovább, és így egy automatákból álló faszerkezetet képeznek. Egy állapotot finomíthatunk több automatává, így valósítható meg az UML-ben használt konkurens régiók működése.

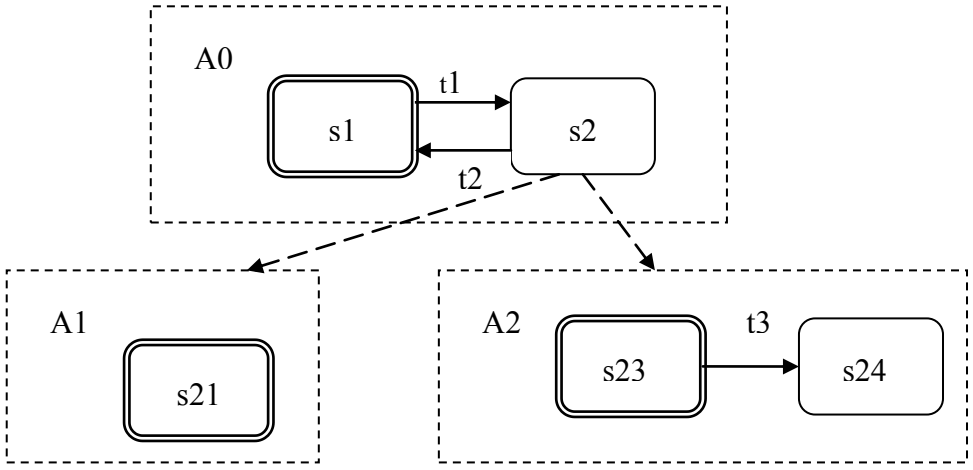
Az EHA tehát a következő hármas: (F, E, ρ), ahol

- F: az EHA-t alkotó szekvenciális automaták (állapotgépek) halmaza,
- E: azon események halmaza, amiket feldolgoz az EHA,
- ρ: finomítási függvény, mely egy szekvenciális automata állapotához szekvenciális automaták egy halmazát rendeli.

A 9. ábrán láthatunk egy állapottérképet, a 10. ábrán pedig az EHA leírását.



9. ábra: Állapottérkép



10. ábra: EHA leírás

Az s2 állapothoz tartoznak alállapotok, így ahhoz az EHA-ban tartoznak finomítások (a szaggatott nyilak jelzik). Mivel konkurens (AND típusú) finomítás történt az állapottérképen, ezért több szekvenciális automata tartozik hozzá az EHA-ban. Az átírás megszünteti a nehezen kezelhető szintek közötti átmeneteket, például a t2 is felkerült az A0 automatába. Így viszont nem az eredeti állapottérkép átírása lenne az EHA, ezért az átmenetek címkézését bővítjük egy forrás- és egy célkijelölés mezővel, melyek tartalmazzák a szükséges információkat, hogy az állapottérkép bármely átmenetét át lehessen transzformálni az EHA-s leírásba. Jelen példákban t2 forráskijelölése {s21}, hisz az eredeti állapottérképben s21-ből indult. Célkijelölése a t1 átmenetnek nem üres, hisz tüzelése után nem csak az EHA s2 állapotába, hanem s21 és s23 állapotába is be kell lépnünk, tehát a célkijelölés az {s21, s23} halmaz.

A teljes transzformáció a következő alfejezetekben leírt lépésekből áll.

3.3.1 UML állapottérkép megadása

A rendszer állapottérképének megtervezésekor a következő konvenciókat kell figyelembe venni, hogy a transzformáció a jelenlegi implementációjában működjön. Osztályszinten lehet definiálni, hogy milyen típusú (FIFO, halmaz) és méretű eseménysorral rendelkezzenek az objektumok. Fel kell venni egy kollaborációs diagramra a rendszerben szereplő objektumokat, és a közöttük lévő kapcsolatokat asszociációk segítségével ábrázolni kell.

A transzformáció későbbi lépései zárt rendszereket tudnak kezelni. Ha az állapottérképünk nem ilyen, akkor modellezzük egy objektummal a külvilágot, és ennek trigger feltétel nélküli átmenetei szolgáltathatják a többi objektumnak az eseményeket.

A jelenlegi implementáció a Rational Rose 2001-ből exportált, XMI (XML Metadata Interchange) szabványnak megfelelő formátumú UML modellekre lett elkészítve.

3.3.2 Állapottérkép → EHA transzformáció

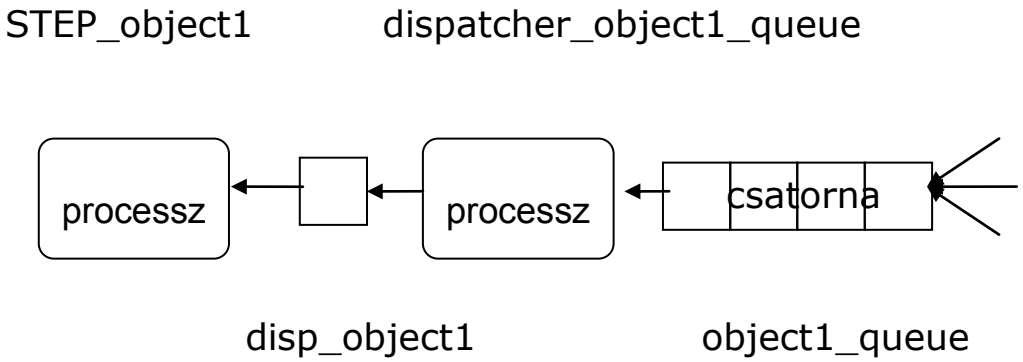
Az állapottérképet a következő lépésben kiterjesztett hierarchikus automatává transzformáljuk egy, a tanszéken készült Prolog program (sc2eha) segítségével. A hierarchikus automata működési szemantikája már jól definiált, azt különböző formátumokba továbbtranszformálni már egyértelmű művelet. A transzformáció eredményeképpen egy XML fájlt kapunk, benne az EHA leírással.

3.3.3 EHA → Promela transzformáció

Az utolsó transzformációs lépésként az EHA modellt Promela leírássá alakítja az eha2promela Java-s program. Az eha2promela program lehetőségeinek a tesztgenerálásnál csak egy részét használjuk ki (a program ugyanis képes dinamikusan változó rendszerekhez is Promela leírást generálni).

Tekintsük még át a generált PROMELA kód szerkezetét, hisz később a tesztgeneráláskor az itt szereplő változókat és konstrukciókat használjuk. (A függelék tartalmaz egy rövidített, megfelelően kommentezett példa Promela kódot, melyben a fontosabb részek Promela nyelvű megadása szerepel.)

Minden egyes objektum állapottérképének megfelel egy processz, ami annak a léptetését végzi. Ezen kívül mindegyik objektumhoz tartozik egy eseménysor, ide érkeznek a neki küldött események. Mivel az UML szabvány ezt nem specifikálja, a kellő flexibilitás érdekében egy eseménysor-kezelőt (dispatcher) rendelnek minden objektumhoz, ez felelős a következő esemény kivételéért. Az eseményküldést Promela csatornák segítségével valósítja meg. Egy szinkron, randevú típusú csatorna biztosítja a kommunikációt a dispatcher és az objektum között. Egy másik, konfigurálható méretű csatorna valósítja meg az eseménysort, ezt a dispatcher működését megvalósító processz olvassa. A 11. ábrán megfigyelhetjük ezek elnevezési konvencióit az object1 nevű objektumhoz generált neveken:



11. ábra: A generált csatornák és processzek

A rendszer állapotát globális bit típusú változókban tárolja, minden egyes állapottérkép minden állapotának megfelel egy bit. Az átmenetek tüzelhetőségét (engedélyezett és nincs nála nagyobb prioritású) is egy globális bit jelzi, így az LTL formulák viszonylag egyszerűek lesznek, pusztán ezeknek a biteknek az igaz értékét kell vizsgálni egy adott állapot vagy átmenet lefedéséhez.

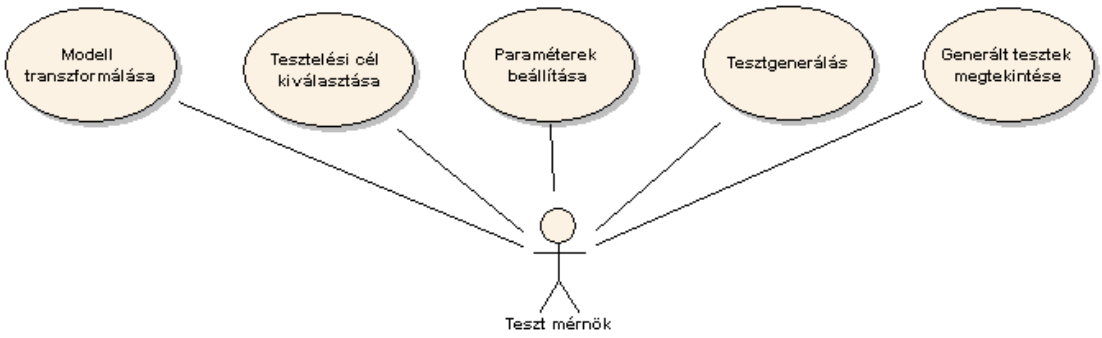
Az egyes állapottérképek léptetését megvalósító processz belsejében először kiválasztja a tüzelhető átmenet halmazokat, ha több tüzelhető halmaz van, akkor azok közül egyet véletlenszerűen eltüzel (köszönhetően a Promela nem-determinisztikus *if* konstrukciójának). A konkurens régiók esetén lehetséges „párhuzamos” végrehajtást új processzek indításával éri el.

4 Tesztgeneráló program implementálása

Elkészítettem egy, a második fejezetben ismertetett tesztgenerálási módszert megvalósító tesztgeneráló alkalmazást, mely az előző fejezetben leírt eszközöket használja és vezérli. Ez a rész az implementáció, egy Java nyelvű program szerkezetét és működését ismerteti.

4.1 Használati esetek

A program működésének főbb lépéseit a 12. ábrán látható használati esetek (use cases) szemléltetik. Ezek fogalmazzák meg a programmal szemben támasztott legfontosabb követelményeket, és egy jó magas szintű áttekintő képet adnak a nyújtott funkcionalitásról.



12. ábra: Használati esetek

4.1.1 Modell megadása

Kiindulás állapot: A tesztgenerálási folyamat elején van a felhasználó.

Használati eset lépései:
1. A tesztelő kijelöli az UML modellből exportált XMI leírást, amihez teszteket szeretne generálni.
2. Megadja, hogy az EHA leírást újragenerálja-e a program az XMI-ból, vagy használjon egy előző futásból meglévőt.
3. Az EHA leírásból Promela leírást generál a rendszer, azt elmenti egy egyedi névvel, mely a modell nevéből és egy sorszámból áll.
4. Az EHA leírásból a memóriában felépít egy struktúrát, mely tartalmazza az állapottérkép összes információját, és azokat a későbbi lépések számára elérhetővé teszi.

Végállapot: az állapottérkép működését megvalósító Promela kód rendelkezésre áll, az állapottérkép információi be vannak töltve a memóriába.

4.1.2 Paraméterek beállítása

Kiindulás állapot: A tesztgenerálást még nem kezdte el a felhasználó.

Használati eset lépései:
1. A felhasználó beállítja a rendszer konfigurációs állományában a tesztgeneráláshoz használt külső eszközök és komponensek elérési útját, valamint megadja, hogy az egyes lépésekről kér-e majd részletes információt.
2. A felhasználó kiválasztja, hogy a modell ellenőrző milyen paraméterezését használja a program a tesztgenerálás során.

Végállapot: A tesztgeneráláshoz szükséges összes külső paraméter beállítva.

4.1.3 Tesztelési cél kiválasztása

Kiindulás állapot: Ha a felhasználó választani akar majd a lehetséges állapotok és átmenetek közül, akkor a „Modell megadása” használati esetnek megfelelően a modell betöltése is szükséges.

Használati eset lépései:
1. A tesztelő kiválasztja az általános tesztelési célt, mely meghatározza, hogy milyen tesztelési követelményekhez kell majd teszteket generálni. Jelenleg a program a „fedési kritérium”, az „adott elemek fedése” és a „saját formula fedése” tesztelési célokat támogatja.
2. Beállítja a kiválasztott tesztelési cél részleteit. Fedési kritérium esetén teljes állapot vagy teljes átmenet fedést választhat. Adott elemek fedése esetén megadhatja, hogy pontosan mely állapotokat vagy mely átmeneteket szeretné lefedni. Saját formula esetén pedig az LTL formulát (a SPIN szintaxisával) és az abban használt kijelentések Promela-beli megfeleltetését kell kifejteni.

Végállapot: Adott a tesztelési cél, ami alapján a program tud tesztelési követelményeket és azokhoz tartozó formulákat generálni.

4.1.4 Tesztgenerálás

Kiindulás állapot: A felhasználó a „Tesztelési cél kiválasztása” használati esetnek megfelelően megadta a tesztelési célokat, és a „Modell megadása” használati esetnek megfelelően betöltötte az EHA leírást.

Használati eset lépései:
1. A program generálja a tesztelési követelményeknek megfelelő LTL formulákat.
2. Felparaméterezi a modell ellenőrzőt a Paraméterek beállítása használati esetben megadottaknak megfelelően.
3. Elvégzi a modell ellenőrzését (leellenőrizteti a formulát).
4. Ha a modell ellenőrző talált ellenpéldát, akkor az azt leíró fájlból kinyeri a teszteset megadásához szükséges eseményeket és akciókat, majd elmenti azokat.

Alternatív lépések:
3b. Túl alacsonyra voltak beállítva az ellenőrzés korlátai, vagy kevés a rendelkezésre álló szabad erőforrás, így a rendszer nem talál tesztet, pedig létezik. A modell ellenőrző jelzi, hogy korlátba ütközött az ellenőrzés során. A tesztgenerálást érdemes megismételni más paraméterekkel.

Végállapot: Ha létezik legalább egy, a követelményt lefedő teszt, akkor a program elment egyet belőlük.

4.1.5 Generált tesztek megtekintése

Kiindulás állapot: Sikeresen lefutott a tesztgenerálás, és legalább egy teszteset elmentett a rendszer.

Használati eset lépései:
1. A felhasználó jelzi, hogy szeretné a teszteket megtekinteni.
2. A program olvasható formában megjeleníti a teszteket.

4.2 Felhasználói felület

A programhoz kétféle, egy parancssoros (konzolos) és egy grafikus felhasználói felület készült. A rendszer belső működésének könnyebb megértése érdekében érdemes ezeket most áttekinteni, hogy tudjuk, az egyes osztályok és hívásaik milyen eredményt produkálnak majd.

4.2.1 Konzolos felület

A program indulásakor a következő paramétereket várja.

- *-xmi*: az exportált XMI fájl teljes elérési útja, a program innentől kezdve ezt a könyvtárat használja munkakönyvtárnak, itt hozza létre a szükséges köztes állományokat és a teszteseteket.
- *-testgoal*: a paraméter értéke a tesztelési célokat határozza meg, ez után kell a kiválasztott cél részleteit megadni. Lehetséges értékei:
    - CoverageCriterion: minden állapot lefedése (allStates) vagy minden átmenet lefedése (allTransitions)
    - SelectedStates, SelectedTransitions: csak a paraméter után pontosvesszővel elválasztva megadott állapot vagy átmenetek lefedése
    - Formula: saját LTL formula, a SPIN szintakszisával megadva.
- *-noeha*: opcionális. Ha meg van adva, nem generál EHA leírást az XMI-ból, hanem a tmp könyvtárban keres egyet.

```
C:\>java -cp .;eha2promela\xerxes.jar testGenerator.consoleClient.TestGeneratorConsole

Usage: TestgeneratorConsole -xmi modell_path -testgoal goal goal_parameter [-noeha]
 -xmi      : path of the XMI modell to transform
 -testgoal : possible values: CoverageCriterion, Formula, SelectedStates or SelectedTransitions
 parameters: CoverageCriterion : allStates or allTransitions
             Formula : LTL formula in Spin syntax
             Selected : state or transition names separated by ;
 -noeha    : if present, the sc2eha_rose will not be invoked
```

13. ábra: Bemeneti paraméterek

A többi beállítást a program a class fájlok testGenerator könyvtárában lévő configuration.xml-ből veszi, ezt a fájlt kell kézzel módosítani, ha például más SPIN paramétereket akarunk használni.

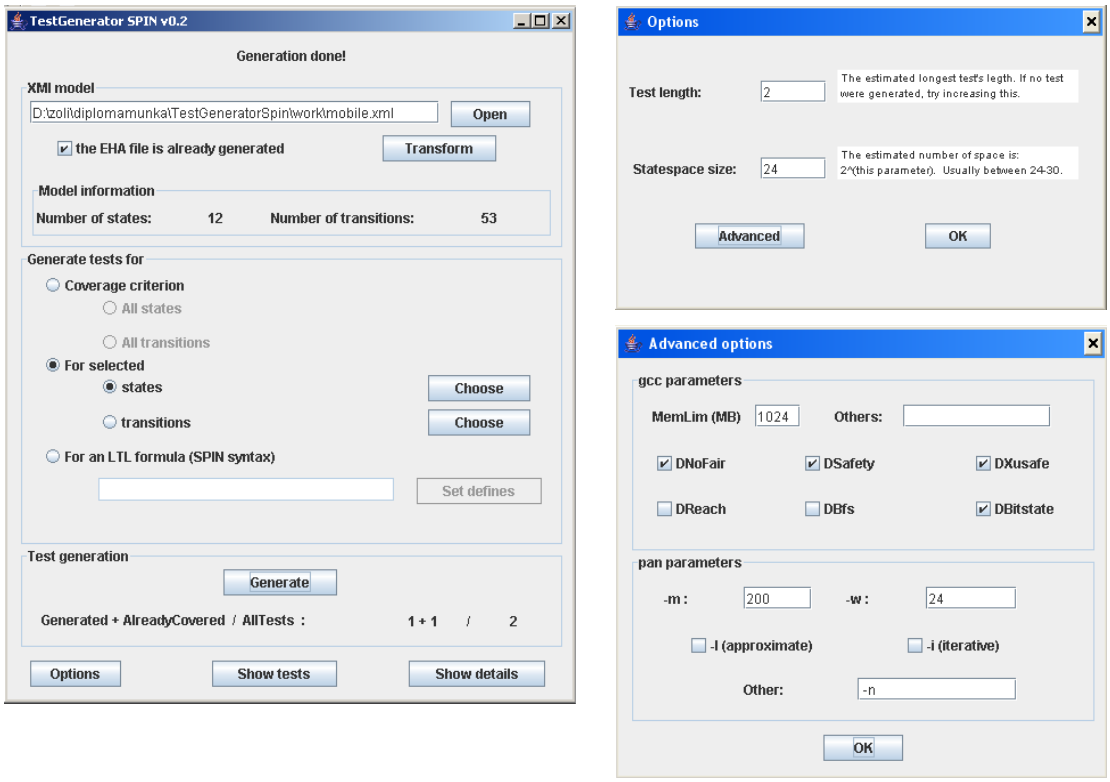
A tesztek megjelenítéséhez a testGenerator/guiClient könyvtárban lévő testcase2html.xsl XSL transzformációval kell a kérdéses tesztesetet transzformálni.

4.2.2 Grafikus felület

A program grafikus felülete a Swing grafikus könyvtárat használja. A 14. ábrán láthatóak a program által használt űrlapok.

A konzolos felülethez képest a grafikus felületen az egyes lépések jobban el vannak választva, azokat a felhasználó indítja el. A gazdagabb felhasználói felületnek köszönhetően itt már nem kell tudni az állapotok vagy átmenetek nevét, ha adott elemekhez szeretnénk teszteket generálni, hanem az a modell beolvasása után a program által felkínált listákból választható.

A SPIN paraméterezésével kapcsolatos beállításokat nem kell külön a configuration.xml-ben szerkeszteni, erre a grafikus felület is lehetőséget biztosít. Az egyszerűbb felület beállításához nem kell ismernünk a SPIN működését, csak azt kell megjósolnunk, hogy milyen hosszúak lesznek maximálisan a tesztjeink, és körülbelül mekkora a rendszer állapottere (ez a paraméter nem kritikus, helytelen megadása esetén is legfeljebb csak lassabb lesz a generálás). Természetesen lehetőség van közvetlenül a SPIN paramétereit is beállítani.

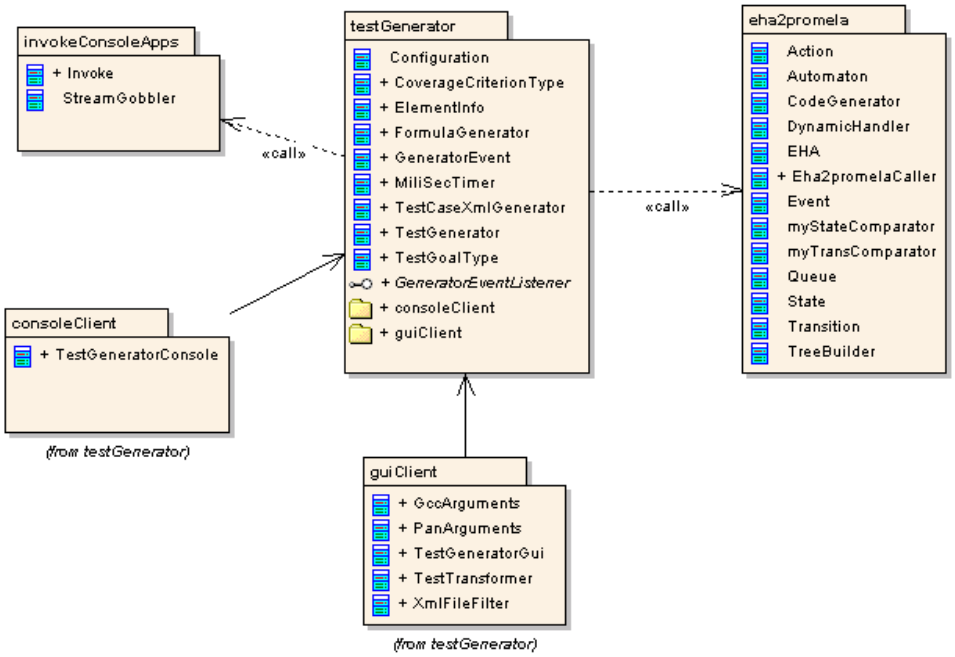


14. ábra: Grafikus felület

A legfontosabb SPIN beállítások külön vezérlők segítségével adhatók meg, a maradék beállítás pedig egy szöveg konstansként, melyet a tesztgeneráló program egy az egyben továbbad a modell ellenőrzőnek.

4.3 Logikai modell

A 15. ábrán a program (némileg rövidített) osztálydiagramja látható, az egyes csomagokban lévő osztályokkal és a közöttük lévő relációkkal.



15. ábra: TestGenerator osztálydiagram

A program logikáját megvalósító osztályok a testGenerator csomagban találhatóak. Hogy könnyű legyen többféle felhasználói felületet és interfészt készíteni a generátor programhoz, a testGenerator csomagban lévő osztályok nincsenek kapcsolatban a felhasználóval. A paraméterek beolvasását és az indítást a kétféle felületet megvalósító consoleClient és guiClient csomag osztályai intézik. Így akár egy másik Java programból is meghívható, könnyedén beépíthető az általa nyújtott funkcionalitás.

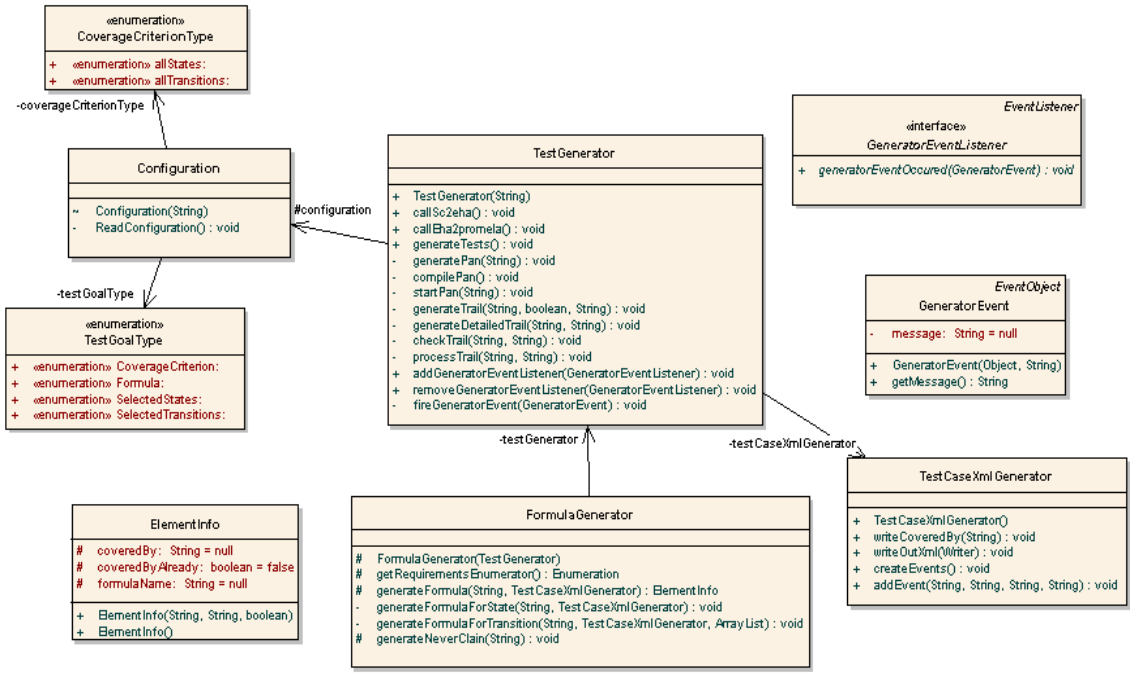
Az invokeConsoleApps csomag osztályai parancssori programok meghívását és az általuk kezelt kimeneti és bemeneti folyamok kezelését teszik egyszerűvé. A SPIN működésének során több lépésben is parancssori programok megfelelő paraméterezését és elindítását igényli, ezért volt szükség erre a funkcióra.

A harmadik fejezetben bemutatott állapottérkép → Promela transzformátor első lépése Prolog programként van megvalósítva, ezt egy batch fájl segítségével hívja meg a program. A második része ugyan Java programként volt megvalósítva, de közvetlenül ezt sem tudtam meghívni, mivel a felhasználói felülettel túlzottan integrálva volt a tényleges belső logika. Ezért készült az Eha2PromelaCaller osztály, ami átadja a megfelelő paramétereket, és legeneráltatja a kívánt Promela állományt. További haszna ennek az osztálynak, hogy a kódgenerálás során az eha2promela program feldolgozza az EHA leírást, így annak az egyes állapotait, átmeneteit már a program adatszerkezeteiből tudom kiolvasni, és nem kell az XML fájlt még egyszer elemezni.

A consoleClient csomag segítségével parancssori felületen adhatjuk meg a tesztgenerálás paramétereit, így a tesztgenerálás például szkriptekből is meghívható. A modell ellenőrző beállításának optimalizálásakor hasznos volt ez a lehetőség, hisz így könnyedén lehetett nagyszámú mérést elvégezni vele.

A guiClient csomag Java Swing felületet biztosít a tesztgeneráláshoz, melyet a TestGeneratorGui osztály valósít meg. A paraméterek kezelését és a SPIN által várt formába való átalakítást végzik a GccArguments és PanArguments osztályok. A TestTransformer feladata pedig a generált tesztek könnyen olvasható formában való megjelenítése, ehhez XSLT transzformációt alkalmaz a háttérben.

A 16. ábra bemutatja a testGenerator csomag részletes felépítését. Az áttekinthetőség kedvéért a diagram nem tartalmazza az osztályok tulajdonságait, valamint a getter és setter metódusokat (A CD mellékleten a Documentation\UmlModel\images könyvtárban megtalálható az összes itt szereplő UML diagram png formátumban is).



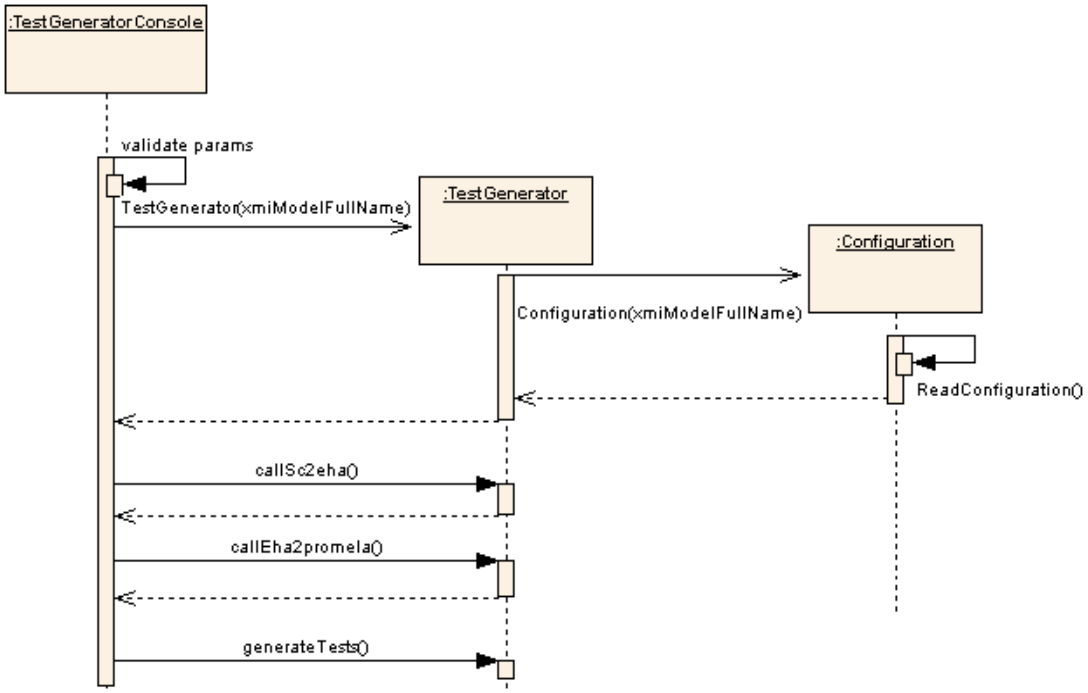
16. ábra: testGenerator csomag osztálydiagramja

A MiliSecTimer, mint a neve is mutatja milliszekundum alapú időmérést tesz lehetővé, erre az optimalizációs mérések kapcsán volt szükségem. A program az összes beállítási lehetőségét egy külső, XML formátumú állományból olvassa be, ennek a feldolgozását és a benne tárolt paraméterek elérését valósítja meg a Configuration osztály.

A teszt követelmények megfogalmazására szolgálnak a TestGoalType és CoverageCriterionType enumerációk. Ha kiválasztottunk egy tesztelési célt a TestGoalType lehetőségei közül, akkor az ehhez szükséges LTL formulák előállítását, és ezeknek a SPIN által használt átalakítását a FormulaGenerator osztály végzi el.

A generálási folyamat állapotáról szóló információkat a TestGenerator osztály a Java eseménykuldése segítségével közvetíti az aktuális, grafikus vagy konzolos kliensének. A kliensek implementálják a GeneratorEventListener interfészt, és a folyamat minden egyes fontos lépése után a TestGenerator a fireGeneratorEvent metódusa segítségével értesít erről mindenkit, aki korábban feliratkozott a listájára az addGeneratorEventListener metódussal.

A program indulását a 17. ábrán lévő szekvencia diagram szemlélteti (az ábrán a konzolos kliens szerepel, de grafikus felület esetén is lényegében ugyanez a hívási sorozat játszódik le).

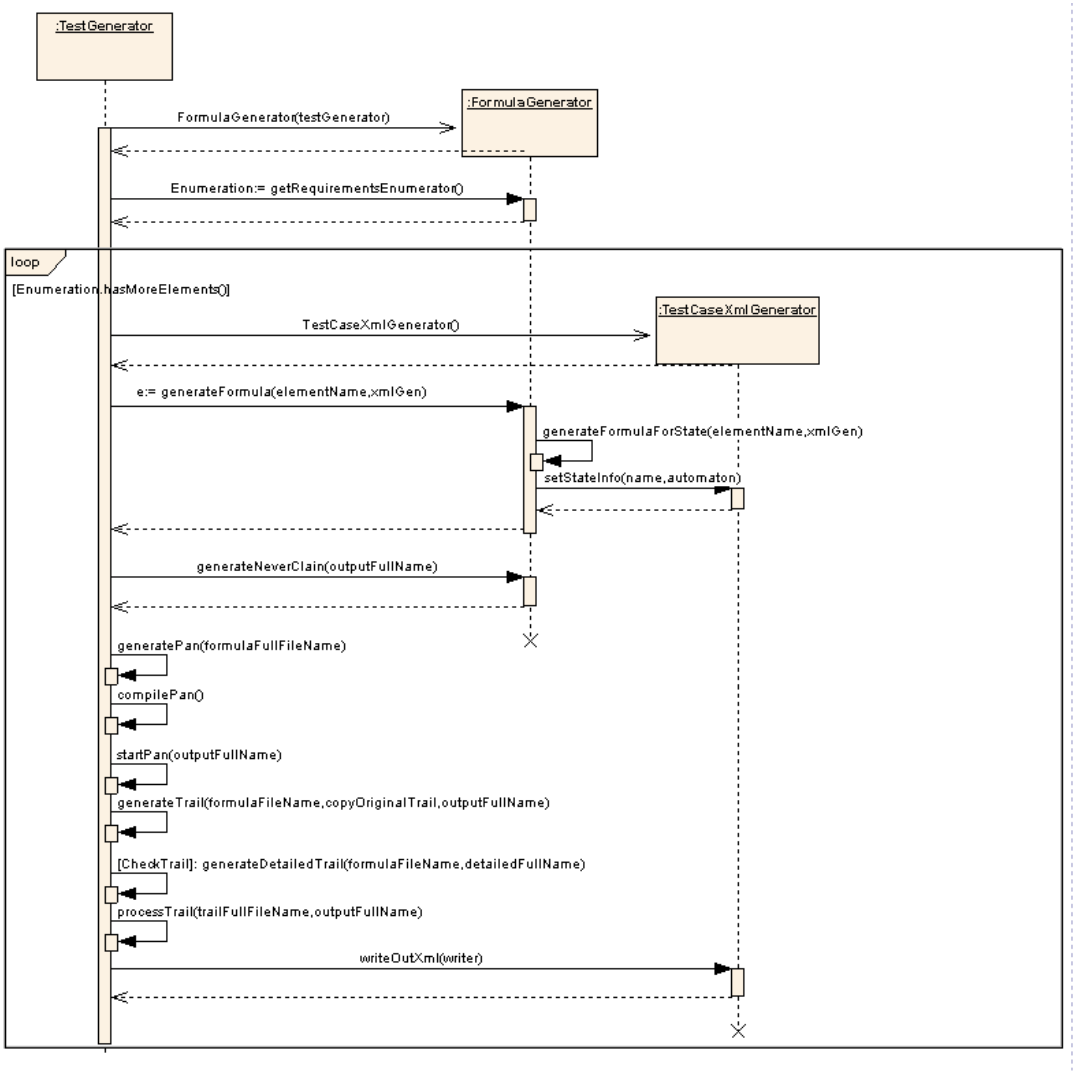


17. ábra: A program indulása

A bemeneti paraméterek ellenőrzése után beolvastatja a konfigurációs beállításokat, majd

- elindítja az állapottérkép → EHA transzformációt,
- elvégzi az EHA → Promela transzformációt,
- végül következik a tesztgenerálás.

A GenerateTests metódus lépéseit érdemes még megvizsgálni.



18. ábra: GenerateTests metódus

A metódus lényege a 18. ábrán látható lépéseket végző ciklus, mely a megadott fedési kritériumtól függően az állapotokon, átmeneteken vagy egy megadott LTL formulán megy végig. Az ábrán állapothoz tartozó tesztek generálását látjuk (a FormulaGenerator generateFormulaForState metódusa hívódik meg). Az áttekinthetőség kedvéért az egyéb, kevésbé fontos hívások nincsenek feltüntetve a diagramon (pl. konfigurációs értékek beolvasása, Invoke osztály Execute metódusának hívása, stb.).

1. **Formula generálása**: A FormulaGenerator osztály a második fejezetben leírtaknak megfelelően generálja a PLTL formulát. A felületen megadott tesztelési cél (TestGoalType) és annak paraméterei (pl. fedés esetén milyen típusú fedés, csak a kiválasztott átmenetek esetén melyek ezek) alapján előállítja azokat az elemeket, amiket le kell fedni a generálás során. Beírja a készülő teszt XML fájlba az elem azonosító információit, majd never claimet generáltat a SPIN-nel a PLTL formulából.
A SPIN formulában csak globális változókra lehet hivatkozni, ez szerencsére nem gond, ugyanis a generált Promela kódban a szükséges információk elérhetők ily módon is.

Teljes állapotfedés esetén generált formulák szerkezete:

```
#define in_[állapot] ( [objektum]_[állapot] == 1 )
! <> (in_[állapot])
```

A <> jel a SPIN-ben az F temporális operátor megfelelője, az [állapot] helyére az aktuális állapot, az [objektum] helyére az aktuális objektum neve helyettesítődik be. Ezeket az információkat az Eha2PromelaCaller osztály metódusai segítségével határozzuk meg. Ez a Promela kód generálása során beolvassa az EHA leírást tartalmazó XML fájlt, és felépít belőle egy „EHA – Automaták – Állapotok, átmenetek" hierarchiájú struktúrát, az egyes elemeknek osztályokat feleltetve meg. A getStateNames metódus végigiterál rajta, és kigyűjti az összes állapot nevét, továbbá, hogy melyik automatába tartozik (ez helyettesítődik be később az [objektum] rész helyébe).
Teljes átmenetfedés esetén pedig a következő alakúak a formulák:

```
#define firing_[tranzíció] ( Cand_[tranzíció] == 1 )
! <> (firing_[tranzíció])
```

Az átmenetek az EHA transzformáció során kapnak egy azonosítót, mivel általában nincs egyértelmű nevük. A program az EHA leírásból a getTransitionIDs metódus segítségével kérdezi le ezeket, és a továbbiakban mindig ezzel hivatkozik rájuk. A getTransitions pedig az átmenet összes leíró információját átadja. Erre a végleges tesztfájl generálására során van szükség, hogy meg tudja jelölni, hogy melyik átmenetről van szó (lásd a covering elem az XML-ben).

3. **GeneratePan metódus**: A következő lépés a pan ellenőrző forrásfájljainak generálása. Itt nincs sok paraméterezési lehetőség, a never claimet tartalmazó formula fájl és a Promela modellt tartalmazó fájl nevét kell megadni.

4. **CompilePan metódus**: A gcc fordító meghívásával a keletkezett C nyelvű állományokat fordítja le. Itt paraméterként felhasználja a metódus a konfigurációs állományban vagy a grafikus felületen megadott argumentumokat.

5. **StartPan metódus**: Itt történik a formula ellenőrzése. A háttérben elindítja a lefordított pan binárist, és megvárja, hogy befejezze a működést, majd elmenti a kimenetét. Ha a modell ellenőrző talál ellenpéldát, akkor egy úgynevezett trail fájlt készít, benne azokkal a lépéssekkel, amik az ellenpéldához vezetnek. Ezen kívül a verifikáció általános információit (memóriaigény, felhasznált adatszerkezetek nagysága, stb.) is megjeleníti, melyeket a metódus külön fájlba ment. A pan-nak is lehet paramétereket megadni (például mélységkorlát a bejárás során), melyek jelentősen befolyásolják a futás időigényét és eredményét, ezekkel a következő fejezet foglalkozik részletesen.

6. **GenerateTrail metódus**: A generált trail fájl a SPIN által használt belső azonosítókat tartalmazza, így azzal nem sok mindent tudnánk kezdeni, ezért a megfelelő kapcsolók segítségével kiolvastatjuk belőle a futás során történt csatorna olvasásokat és írásokat, majd ezeket elmentjük egy külön állományba.

7. **CheckTrail metódus:** Ez a metódus akkor fut csak le, ha a konfigurációs állományban engedélyezve van, a generált teszteket vizsgálata abból a szempontból, hogy milyen más követelményeket fednek le. Ilyenkor részletes trail fájlt generáltat, melyben minden végrehajtott lépéshez tartozó kód szerepel, és a lépések után kiírja a

SPIN, hogy mi az aktuális állapotkonfiguráció. Ebben az egyébként elég nagy méretű fájlban kell Cand_ kezdetű és =1 végű valamint =1 végű és valamelyik állapot nevével kezdődő sorokat keresni. Az első típusú sorok jelzik az egyes átmenetek tüzelését, a második típusúak pedig az állapotba való belépéseket.

8. **ProcessTrail metódus**: Ez a metódus állítja elő a teszteseteket az esemény küldésekből és fogadásokból. A fő problémát itt is az okozta, hogy a SPIN kimenetét emberi olvasásra szánták, nem pedig gépi feldolgozásra, így viszonylag nehézkes a szükséges információ automatikus kinyerése. Megfelelő mintákat kell keresni a sorokban (pl. a végén zárójelben lévő szó disp_ karakterekkel kezdődik, és a sorban van egy -> irányú nyíl), és ebből lehet megállapítani, hogy ki kinek küldött eseményeket. Azért, hogy később könnyebb legyen feldolgozni vagy megjeleníteni az eredményt, a metódus az előállított esemény és akció sorozatból egy XML fájlt generál, melyre a függelékben láthatunk egy példát. A generált teszteseteket jelenleg egy XSL transzformáció segítségével HTML formátumúvá lehet alakítani, hogy könnyebb legyen áttekinteni.

4.4 Java implementáció

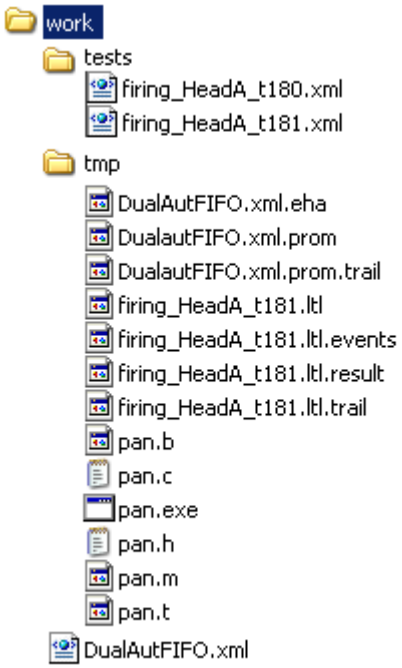
Az elkészült Java nyelvű implementáció legfontosabb adatai:

- Java 2 Standard Edtion 1.5 futtatókörnyezetet használ.
- 18 darab osztály (az eha2promela komponenssel együtt 30), 256 darab metódus.
- ~4500 sor kód (eha2promela nélkül, kommentekkel).
- 217 KB forráskód.

A forráskód részei a *javadoc* konvencióknak megfelelően lettek kommentezve, így a további implementáció-specifikus információk megtalálhatóak a kommentekből generált dokumentációban a CD mellékleten.

4.5 A program által generált fájlok

A generált fájltípusok láthatóak a 19. ábrán. A kiinduló fájl ebben az esetben a DualAutFIFO.xml, ez a modell leírása. Ennek a könyvtárában létrehoz a program egy tmp nevű alkönyvtárat, ide menti el a tesztgenerátor az összes keletkező ideiglenes fájlt, így egy esetleges hibakeresés könnyebben megvalósítható. Először az eha és prom kiterjesztésűek generálódnak, majd ahogy elindul a tesztgenerálás, mindegyik állapothoz (átmenethez) keletkezik négy darab fájl.



19. ábra: A generált fájlok

- *.ltl: a generált never claim, szerepel benne kommentként az LTL formula is.
- *.trail: az ellenőrzés során kapott trail fájl. Ez alapján később akár a SPIN-ben is meg lehet tekinteni ezt a futást irányított szimuláció segítségével.
- *. result: a verifikáció eredménye. Akkor lehet fontos, ha valamiért nem sikerült tesztet generálni, ilyenkor ebben megtaláljuk az okát, például kevés volt a memória vagy túl kicsi volt a megadott mélységkorlát az állapottér bejárása során.
- *.events: a trail fájlokban szereplő küldések és fogadások szöveges formában.

A könyvtárban található pan kezdetű fájlok a legutolsó verifikáció ideiglenes állományai.

A tests könyvtárba kerülnek a teszteset formába konvertált ellenpéldák, minden teszt kritériumhoz (jelen esetben minden állapothoz vagy átmenethez) egy darab.

5 Optimalizálás

A tesztgenerálás során alapvetően más módon szeretnénk használni a modell ellenőrző eszközt, mint a klasszikus verifikációban. Ott főleg biztonsági és élőségi tulajdonságokat ellenőrzünk a teljes állapottér bejárásával. Az a megnyugtató eredmény, ha az eszköz jelzi, hogy minden lehetséges futást megvizsgált, és mindig teljesültek a feltételeink (pl. nem kerültünk veszélyes állapotba, mindenhonnan el lehet érni a kezdőállapotot, stb.), ezzel tudjuk formálisan bizonyítani a rendszer biztonságosságát. A tesztgenerálás során a cél az, hogy a modell ellenőrző találjon egy ellenpéldát, lehetőleg minél rövidebbet, minél rövidebb idő alatt.

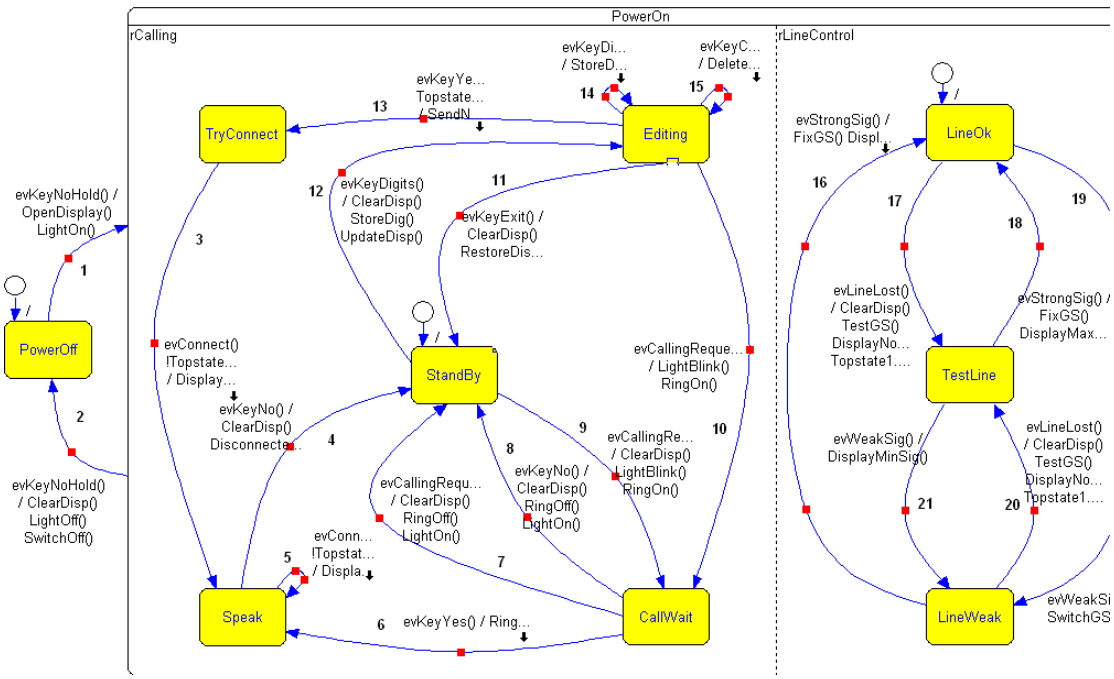
Ezért az elkészült SPIN modellellenőrzőt felhasználó tesztgenerátoron méréseket végeztem, hogy meghatározzam azt, hogy

- a modell ellenőrző eszköz különböző paraméterei hogyan befolyásolják a futási időt,
- az egyes lépések mekkora részét képezik a teljes folyamatnak,
- végül összehasonlítottam, hogy ha ugyanehhez a példához az SMV modellellenőrző eszköz segítségével generálnánk teszteket, akkor milyen eredményeket kapnánk.

Mérési környezet:
A mérés során használt eszközök:

- Hardver: Celeron 2GHz, 512 MB RAM.
- Szoftver: Windows XP Prof SP1, SPIN 4.1.3 Windows verzió, SWI Prolog 5, Java 1.4.2.
- Modell: az IAR visualSTATE programban található mobil telefon állapottérkép példát használtam bemeneti modellként.

Az állapottérkép a 20. ábrán látható, 9 darab állapotot és 21 darab átmenetet tartalmaz, a program a futás során ezekhez generált egy-egy tesztesetet.



20. ábra: Mobile mintapélda

5.1 Tesztgenerálás SPIN modell ellenőrzővel

5.1.1 Az egyes lépések időigénye

Az XMI → EHA transzformáció időigénye átlagosan 5,4s volt.
Az EHA → Promela transzformáció időigénye 1,3s – 1,6s között változott.
Ezután méréseket végeztem különböző SPIN paraméterekkel, hogy a folyamat egyes részei mekkora részét képezik a tesztgenerálásnak:

2. táblázat: Részlépések által felhasznált idő

Kapcsolók	*formula*	*Generate Pan*	*Compile Pan*	*runPan*	*Generate Test*
Alapértelmezett beállítások	0,249s	0,704s	44,919s	75,01s	0,704s
Mélységkorlát megadva	0,249s	0,673s	43,73s	73,90s	0,673s
Mélységkorlát és DREACH kapcsoló	0,264s	0,657s	44,11s	192,48s	0,657s
Mélységkorlát és rövid futások keresése beállításokkal	0,232s	0,688s	44,374s	553,51s	0,688s

Látható, hogy a formula, a pan forrásfájlok, a fordítás és a tesztfájlok generálása nagyjából konstans idejűek, függetlenek a SPIN-nek megadott opcióktól. A fordítással töltött idő érdemel ezek közül figyelmet, a rövid futások esetén a teljes idő 35%-át tölti fordítással.
A pan futtatása, a tényleges ellenőrzés lefutása pedig jelentősen függ a megadott paraméterektől. A következőkben ezeket vizsgáltam meg.

5.1.2 A gcc argumentumainak hatása

A változtatott paraméterek:
- DMEMLIM=N : a pan N MB memóriát használhat
- DNOFAIR : weak fairness (gyenge méltányosság) ellenőrzésének letiltása
- DSAFETY : ciklusok észlelését letiltja
- DXUSAFE : x[rs] (csatorna kizárólagos használata) asszertációkat nem ellenőriz
- DREACH : -i és –I (iterált módon rövidebb futások keresése) helyes lefutásához szükséges

A mérés során használt beállítások:
- A mérések során a NOEHA paraméterrel indítottam a tesztgenerátort, így az EHA fájlt nem generálta le, hanem a meglévővel dolgozott.
- Az EHA → Promela transzformáció ideje: 1,636s (10 futásból átlagolva)
- Állapotfedésnek megfelelő teszteket generáltam.

A méréseket többféle pan futási paraméterrel is elvégeztem, hogy látszódjon, hogy különböző hosszúságú futások esetén mennyit számítanak az egyes opciók. A három

használt opció a –m a mélységkorlát megadása, és a –i és –I, részletesebb leírásuk a következő alfejezetben, a futási idejű kapcsolóknál található.
A kapott futási idők a lentebbi táblázatokban láthatóak.

A táblázatokban az összes eseményszám a generált 12 tesztesetben szereplő események összege, az in_Speak pedig az az állapot, amelyikhez a leghosszabb tüzelési sorozat vezet. Az eredmények jól szemléltetik a nagyságrendi különbségeket.

- Rövid futások keresése nélkül, pan paraméter: –m1000

3. táblázat: Futási idők –m1000 paraméter esetén

gcc kapcsolók	*Futási idő*	*Összes eseményszám*	*in_Speak hossza*
Nincs	2m2.93s	82	9
-DMEMLIM=300	2m2.75s	103	9
-DSAFETY	2m3.86s	102	9
-DNOFAIR	2m2.95s	89	9
-DXUSAFE	2m3.55s	101	9
-DREACH	4m8.37s	88	15
-DREACH -DMEMLIM300 -DNOFAIR -DSAFETY -DXUSAFE	4m3.28s	75	15

- Rövid futás keresése, pan paraméter: –m1000 –I

4. táblázat: Futási idők –m1000 –I paraméterek esetén

gcc kapcsolók	*Futási idő*	*Összes eseményszám*	*in_Speak hossza*
nincs	3m4.259s	23	9
-DMEMLIM=300	3m4.301s	23	9
-DSAFETY	3m0.166s	23	9
-DNOFAIR	3m1.498s	23	9
-DXUSAFE	3m3.512s	23	9
-DREACH	10m59.25s	17	4
-DREACH -DMEMLIM300 -DNOFAIR -DSAFETY -DXUSAFE	11m7.068s	19	4

- Legrövidebb futás keresése, pan paraméterei: –m1000 –i (a DCOLLAPSE memóriatömörítést be kellett kapcsolni, mert 1GB memóriát használt már).

5. táblázat: Futási idők –m1000 –i paraméterek esetén

gcc kapcsolók	*Futási idő*	*Összes eseményszám*	*in_Speak hossza*
-DMEMLIM=300 -DNOFAIR -DSAFETY -DXUSAFE	15m35.61s	41	9
-DREACH –DMEMLIM=300 -DNOFAIR -DSAFETY -DXUSAFE	21m23.12s	23	9

Az eredményekből látszik, hogy ennél a modellnél a legtöbb kapcsolónak nincs mérhető hatása. Egyedül a –DREACH kapcsoló megadása hozott drasztikus eredményeket, ilyenkor hosszabb futás esetén már egy nagyságrendnyi különbség mutatkozott. Viszont a SPIN dokumentáció szerint ez a kapcsoló szükséges, ha a –i vagy –I opciókat akarjuk használni a futtatás során, és az eredményekből látszik is, hogy ilyenkor rövidebb teszteket generált.

Összességében, a DREACH-en kívül a többit érdemes mindig megadni, a DREACH-et pedig csak akkor, ha használjuk a –i vagy –I opciót.

A SPIN fordítási kapcsolói közül külön figyelmet érdemel a –DBFS kapcsoló, ennek hatására szélességi bejárást használ. Az eddig használt elrendezésben ez nem volt használható, túl lassú volt, 80 lépés mélységig körülbelül 60 perc és 255 MB memória elhasználása után jutott el, és a három átmenet tüzeléséhez kb. 120 lépés kell, így az állapotok nagy részére nem talált tesztet.
Azonban a modell kis módosításával már használható volt:

- A csatornák méretét nullára csökkentettem, így teljesen szinkron működésűvé vált a rendszer.
- További optimalizáció lehet, hogy atomic konstrukciókat helyezünk el az eha2promela által generált Promela kódban, az átmenet hatására történő műveleteket lehet így összefogni.

A módosításokra és azok hatására a következő fejezet részletesen kitér, ugyanis a nagyobb állapottérkép tesztelése során ezekhez a megoldásokhoz kellett folyamodni.

6. táblázat: Futási idők szélességi bejárás esetén

Optimalizáció	*Futási idő*	*Összes eseményszám*	*in_Speak hossza*	*Átmenetek száma*
Csatorna mérete 0	11m48.838s	17	3	914931
Csatorna mérete 0 és atomic	0m47.221s	17	3	470506

Látható, hogy így már a szélességi bejárás használata is használható módszerré vált, gyors, és a legrövidebb teszteket generálja.

5.1.3 A pan futtatásakor megadott argumentumok hatása

A változtatott paraméterek:

- –m : az ellenőrzés során futó mélységi bejárásnak lehet mélységi korlátot megadni ennek a paraméternek a segítségével. Érdemes a modell méretének megfelelő korlátot beállítani, mert egy teszteset általában egy rövidebb eseménysorozat. Hajlamos a SPIN arra, hogy tüzelési ciklusokat egymás után többször bejárjon, és teljesen rossz irányba induljon el a teszteset generálásához. Ha meg van adva megfelelő mélységkorlát, akkor, ha el is indul rossz irányba, beleütközik a korlátba, és visszalép, más irányt próbál a bejárásnál. Viszont ha megadunk, és nem talál tesztet –erre a tesztgenerátor program figyelmeztet– akkor valószínű, hogy csak ezt a korlátot kell megnövelni.
- –i és –I: ha talál egy ellenpéldát a SPIN, akkor képes arra, hogy iteratív módon rövidebbeket keressen. Ez néha hosszadalmas, viszont jelentősen lerövidítheti a teszteset hosszát. A –i kapcsoló a lehető legrövidebbet keresi, a –I csak közelítő megoldást ad. A mérésekben használt kis példában az –i kapcsoló megtalálta a lehető legrövidebb teszteket, a –I pedig néhány eseménnyel többet generált. A konkrét feladattól függ, hogy a tesztgenerálásra vagy a tesztelésre fordított idő a drágább, és ennek megfelelően választhatjuk a rövidebb teszteket vagy a gyorsabb generálást.

A következő táblázatban láthatók a futási eredmények állapotfedés esetén:

7. táblázat: Futási idők teljes állapotfedés esetén, SPIN modell ellenőrzővel

pan futtatási kapcsolók	*Futási idő*	*Összes eseményszám*	*in_Speak hossza*
-i -m1000	4m47.235s	17	3
-i	22m32.468s	17	3
-I -m1000	1m46.641s	22	4
-I	2m48.780s	25	6
-m1000	1m25.486s	97	16
paraméter nélkül	2m04.869s	385	94

A 8. táblázatban pedig az átmenet fedés esetén:

8. táblázat: Futási idők teljes átmenetfedés esetén, SPIN modell ellenőrzővel

pan futtatási kapcsolók	*Futási idő*	*firing_t103 hossza (Speak-Speak)*
-i -m1000	26m0.136s	4
-i	297m24.977s	4
-I -m1000	7m36.830s	8
-I	17m44.136s	4
-m1000	4m54.411s	18
paraméter nélkül	9m17.932s	179

Megjegyzés: ez a két mérés még a 4.1.2-es SPIN-nel készült, ezért az eredmények kicsit eltérnek az eddigiektől.

A táblázatokból látszik, hogy ezeknek a kapcsolóknak van igazából nagyságrendbeli hatása a végeredményre. Mélységkorlátot érdemes mindig megadni, jelentősen csökkenti a futási időt, és néha még az események számát is.
A rövidebb tesztek keresésre szolgáló –i és –I megadásával viszont vigyázni kell, jelentősen lassíthat. A mérésekből úgy tűnik, hogy nem érdemes a –i kapcsolót használni, a –I is elég jó eredményeket produkál, viszont sokkal rövidebb idő alatt.

Megjegyzés: a mérések elvégzéséhez a következő eszközöket használtam:
- Unix time parancs.
- A TestGenerator programot kiegészítettem egy MiliSecTimer osztállyal, ami az elindítása és leállítása között eltelt milliszekundomokat méri.
- A generált teszteset hosszát két VBScript (ossz_esemenyszam.vbs, sendInput.vbs) számolta meg.
- Az egyes mérések csak a configuration.xml tartalmában térnek el egymástól, ennek a cseréjét, és utána a program indítását, majd a tesztesetek elmentését egy bash szkript végezte.

5.1.4 A generált tesztesetek optimális beállítások esetén

Az előző alfejezetek mérései után végül a következő beállításokat találtam optimálisnak (ez kis és közepes méretű modellekre vonatkozik, a következő alfejezetben látni fogjuk, hogy nagyméretű modellek esetén más eljárásokat is kell alkalmazni):
- A fordítás során használható kapcsolókat mind megadtam, kivéve a DREACH-et.
- Nem használtam az iteratív módon rövid utakat kereső –i és –I kapcsolókat.
- Helyette a mélységkorlátot kell nagyon jól beállítani. A különböző paraméterekkel végrehajtott mérések azt mutatták, hogy ezzel lehet a legtöbb időt megspórolni (500-ról 200-ra állítva a mobile példán 4 perc 26 másodpercről 46 másodpercre (!) csökkent a végrehajtási idő). Ezen kívül sokkal rövidebb tesztek keletkeztek. Érdemes a tesztgenerálás előtt az előreláthatólag leghosszabb úton elérhető állapottal/átmenettel kicsit próbálkozni, hogy meghatározzuk, hogy a modellben milyen mélységkorláttal kell számolni. Kétszázról indítva, és százasával növelve pár perc alatt eljutunk az aktuális modellnek megfelelő értékhez, azonban ezzel később rengeteget spórolhatunk.

- A never claimek teljes verifikáláshoz az –a kapcsolót szükséges lenne megadni, azonban nekünk erre nincs szükségünk, csak lassít a verifikáción, és általában e nélkül is találunk tesztet.
- A másik nagyon fontos kapcsoló a –w, amivel a keresés során használt hash tábla méretét lehet beállítani. Erről részletesebben a következő fejezetben lesz szó, elöljáróban csak annyit, hogy az alapértelmezett értéke túl kicsi a jelen feladatokhoz, így sok ütközés van a futás során. Ezért az értékét 24-re állítottam, ez 67.109 MB-ot jelent.

A futási idők a 9. táblázatban, a generált tesztesetek pedig a 10. táblázatban láthatók.

9. táblázat: Futási idők optimális esetben

Tesztek hossza	*Futási idő*
Állapotfedés:	0m46.718s
Átmenet fedés	4m19.244s

Az átmenet fedést generáló futás ideje azért sokkal nagyobb, mert ott 300-ra kellett növelni a mélységkorlátot, hogy minden átmenethez találjon tesztet. Ez pedig jelentősen megnöveli a bejárandó állapottér részt.

10. táblázat: A generált tesztesetek

teszteset	*1.*	*2.*	*3.*
CallWait	evKeyNoHold / OpenDisplay	evCallingRequest / RingOn	
Editing	evKeyNoHold / OpenDisplay	evKeyDigits / StoreDigit	
LineOk	evKeyNoHold / OpenDisplay		
LineWeak	evKeyNoHold / OpenDisplay	evWeakSignal /	
PowerOn	evKeyNoHold / OpenDisplay		
Speak	evKeyNoHold / OpenDisplay	evCallingRequest / RingOn	evKeyYes
StandBy	evKeyNoHold / OpenDisplay		
TestLine	evKeyNoHold / OpenDisplay	evLineLost /	
TryConnect	evKeyNoHold / OpenDisplay	evKeyDigits / StoreDigit	evKeyYes

5.2 Tesztgenerálás SMV segítségével

Összehasonlításképp a tesztgenerálást úgy is elvégeztem ezzel a modellel, hogy az SMV modellellenőrzőt használtam. Ehhez nincs kész program, az egyes lépéseket jórészt kézzel, vagy kisebb szkriptek segítségével végeztem el.

Az egyes lépések hossza:

11. táblázat: lépések hossza, SMV modell ellenőrző esetén

Név	*Idő*
SMV kódgenerálás (VBScript)	0m1.496s
Fedési kritérium generálása (VBScript)	0m0.833s
Összes átmenet formula ellenőrzése	0m1.547s
Összes állapot formula ellenőrzése	0m1.727s
Eredmény szűrése egrep segítségével	0m7.113s
Eredmény szétszedése külön fájlokba, input / output forma előállítása	
Összesen	10,98 s

A generált tesztesetek:

12. táblázat: tesztek hossza, SMV modell ellenőrző esetén

Tesztek hossza	*Összes eseményszám*	*in_Speak / firing_t103 hossza*
Állapotfedés:	17	3
Átmenetfedés	59	4

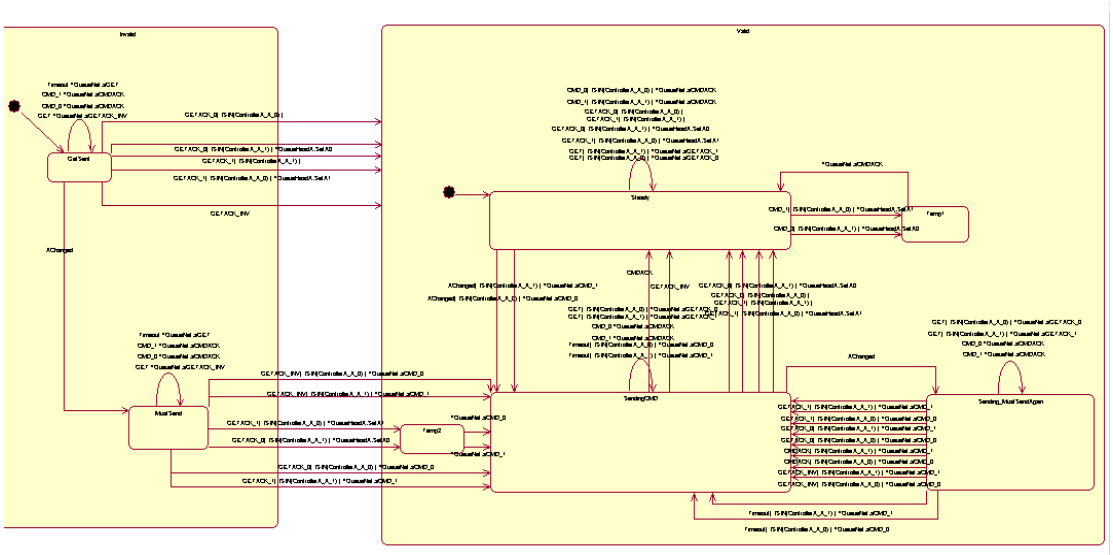
Az SMV-vel végzett mérések esetén sokkal jobb eredményeket kaptam, mint a SPIN használatakor. Ez főképp a következőknek köszönhető:

- Nincs szükség fordításra, egyből ellenőrizhető az ellenpélda. Mivel rövid ellenőrzésekről van szó, ezért nem érvényesülnek a külön ellenőrző fordításával járó optimalizációk.
- Egyetlen futtatással lehetséges több formula ellenőrzése is.
- Nincs szükség iteratív keresésre a rövid ellenpéldák generálásához.
- A Promela kód, az automatikus generálásnak köszönhetően, sokkal általánosabb, míg az SMV kódot csak az ehhez a modellhez szükséges konstrukciókat tartalmazta.

5.3 Ipari mintapélda: szinkronizációs protokoll

A tesztgenerálás alkalmazhatóságát egy valós alkalmazás, egy szinkronizációs protokoll működését leíró állapottérképen teszteltem. A protokoll feladata két számítógép között vezérlőbitek szinkronban tartása. A protokoll valósidejű környezetben működik, az egyes objektumoknak egy-egy task felel meg. A taskok időközönként ellenőrzik, hogy a szinkronban tartandó bitek lokális megfelelőiben történt-e változás. Ha igen, akkor egy üzenetet küld a másik tasknak a kettőjük között lévő dedikált hálózaton, melyben értesíti erről. A másik az üzenet vétele után a hozzá tartozó vezérlőt utasítja a saját bitjeinek a megváltoztatására. A protokoll ezen kívül az üzenetekben nyilván tartja még, hogy a 16 lehetséges útvonal közül melyiken kell azt elküldeni, azonban ezt a modellezés során elhagyták. Ugyanis az ellenőrzés során a hangsúly azon volt, hogy mindig elküldi-e az üzenetet a szinkronizációról, és ez az egyszerűsítés jelentősen csökkentette, az amúgy is nagy modell méretét.

A 21. ábrán a protokoll állapottérkép modelljének egy részlete látható, az egyik félhez tartozó állapotok. Ezen kívül még két vezérlő és a köztük lévő hálózat volt modellezve az állapottérképeken (az ábra szerepe pusztán csak annyi, hogy szemléltesse a modell méretét).



21. ábra: Szinkronizációs protokoll

Az állapottérkép méretét jelzik a következő adatok:

- 5 darab objektum, mindegyik külön állapottérképpel,
- összesen 31 darab állapot,
- összesen 174 darab átmenet.

A generált Promela kódon már valami egyszerűbb tulajdonság ellenőrzése is komolyabb verifikációs feladatnak számít. Az állapotvektor, ami azokat a változóértékeket tartalmazza, amik szükségesek egy állapot azonosításához az állapottér bejárása során, 216 byte-os volt. A mérések során nem volt ritka, hogy akár 2e+08 állapotot járt be a modell ellenőrző, tehát közel 40 GB memória kellett volna ennek az eltárolására. Ebből következik, hogy valamilyen tömörítési technikát kell alkalmaznunk, ha valós példákra is szeretnénk alkalmazni a tesztgeneráló alkalmazást.

A SPIN több ilyen lehetőséget kínál:

- *COLLAPSE* mód: közepes hatékonyságú tömörítés, de ez is kevésnek bizonyult, ráadásul jelentősen megnöveli a futási időt.
- *HC4* mód: hash-compact tömörítési eljárás, a legnagyobb hatékonyságú, így a leglassabb is.
- *BITSTATE* mód: közelítő megoldás, egy bit méretű hash táblát használ.

A továbbiakban a BITSTATE mód [Holz03] lesz fontos számunkra, ezért vizsgáljuk meg azt. Az automata alapú módszer esetén a modell ellenőrzési problémát egy elérhetőségi problémára vezettük vissza, alapesetben ezt a SPIN egy mélységi bejárással oldja meg. Egy hash táblában eltárolja azokat az állapotokat, amiket már bejárt, hogy azokat még egyszer már ne vizsgálja meg (ennek a hash táblának a méretét lehet a –w futásidejű kapcsolóval szabályozni). Akkor választottuk meg jól ennek a méretét, ha közel azonos a ténylegesen bejárt különböző állapotok számával. Ha ennél kisebb lenne, akkor lassú lenne a verifikáció, mert az azonos hash kódú elemeket a tábla adott bejegyzésénél egy láncolt listában tárolja, míg ha a hash tábla mérete túl nagy, akkor feleslegesen foglaltunk neki memóriát és lassabb is lesz a hash érték képzése.

Bitstate hashing esetén a következőt teszi a SPIN. A hash tábla egy bejegyzésének a méretét 1 bitre állítja[1], így ha ütközés történik (azaz ugyanarra a hash kódra képződne le két különböző állapot), akkor a második állapotot is úgy veszi, mintha bejárta volna, ezért ez csak közelítő módszer. Viszont ha egy ellenpéldát talál, akkor az természetesen egy jó ellenpélda, és a tesztgenerálás során nekünk ez a fontos. Remélhetőleg tucatnyi ellenpélda van arra az állításra, hogy az egyik állapotba nem lehet eljutni, azonban számunkra az a lényeges, hogy legalább egyet találjunk. A bitstate hashing módszerrel viszont rengeteg memóriát lehet spórolni, hisz a szinkronizációs protokoll példa esetén a 216 byte-os állapotvektor helyett pusztán egy bitet kell tárolni, így ennek a modellnek az ellenőrzése is lehetővé válik.

A verifikáció még így is túl lassú volt, úgyhogy további módszereket kellett bevetni a komplexitás csökkentése érdekében:

- A kommunikációhoz használt csatornák méretét csökkentettem az alapértelmezett tízről amennyire lehetett (a modellben a kezdőállapotban is kellett két üzenetnek lennie bizonyos csatornákban, hogy a protokoll elinduljon). A legszerencsésebb, ha a 0 méretűre lehet csökkenteni a csatornákat, ekkor ugyanis szinkron működésűvé válnak. Azaz addig nem tudnak új üzenetet küldeni a csatornába, amíg az előzőt ki nem vette, és fel nem dolgozta valaki. Így sokkal kevesebb az egymástól független, tetszőleges sorrendben végrehajtható művelet, ami jelentősen csökkenti az állapottér nagyságát.
- Atomic konstrukció alkalmazása ahol csak lehetséges, például a dispatcher kódjában, csatorna kizárólagos használatát jelző xs és xr kulcsszavak beillesztése (ezek nem hoztak olyan jelentős javulást, főleg a csatorna méretének csökkentéséhez képest).

A fentebbi módosítások jelentősen megváltoztatják a működést, lehet olyan állapot, hogy a szűkítéssel már nem találunk hozzá tesztet. Tekintsük például a 22. ábrán lévő két állapottérképet! A felső az A objektum, az alsó a B-ét írja le. Tegyük fel, hogy a B-hez tartozó eseménysor kettő hosszú, ekkor lehetséges a következő tüzelési szekvencia:

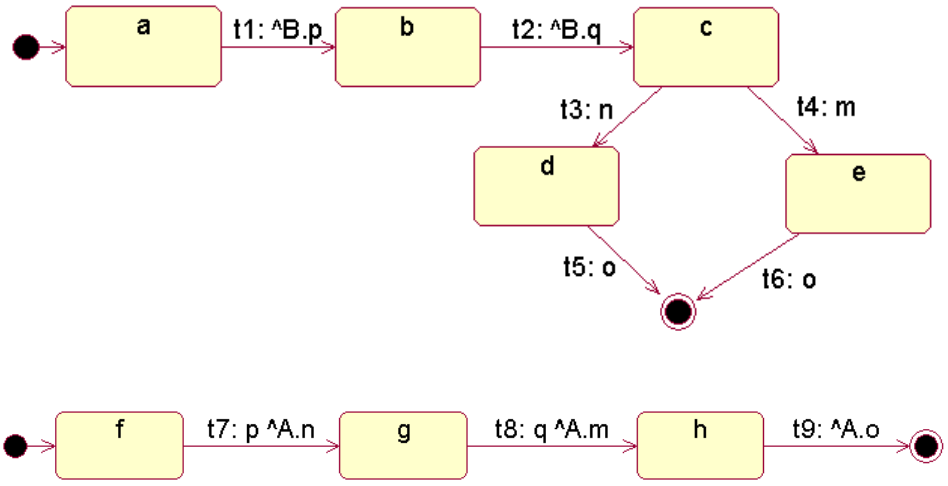
[1] A 2004. októberben megjelent új SPIN verzióban már alapértelmezésként két bitet használ, ugyanis kétféle hash-függvénnyel is kiszámolja az értéket, így csökkentve az ütközés valószínűségét, és ezzel növelve az állapottér lefedettségét.

	Kezdő állapot	*t1 tüzel*	*t2 tüzel*	*t7 tüzel*	*t3 tüzel*
A állapota	a	b	c	c	d
B állapota	f	f	f	g	g

Ez a tüzelési sorozat a d állapot egy lehetséges tesztje. Próbáljunk meg ugyanehhez az állapothoz tesztet találni, ha a B eseménysora nulla nagyságú!

	Kezdő állapot	*t1 tüzel*	*t7 tüzel*	*t2 tüzel*	*t8 tüzel*	*t4 tüzel*
A állapota	a	b	b	c	c	e
B állapota	f	f	g	g	h	h

t1 tüzelés után csak t7-t tudjuk teljesen tüzelni, hisz t2 újabb eseményt rakna B eseménysorába, és alapértelmezésben ilyenkor a SPIN blokkolja a küldő processzt. t7 n eseményt küld A-nak, azonban a b állapotban azt nem tudja feldolgozni, így eldobja azt. Csak t2-t tudjuk tüzelni, utána t8-at, majd t4 tüzelésével az e állapotba jutunk, kihagyjuk d-t, innen nem tudunk átmenni. Láthatjuk tehát, hogy az eseménysor méretének csökkentésével tesztet veszíthetünk, azaz lehet olyan állapot, amihez nem tudunk tesztet generálni.



22. ábra: Szűkítések állapottérképeken

Azonban ami a szűkített rendszerben ellenpélda, az ellenpélda az eredetiben is, hiszen ott is végre lehet hajtani ugyanezt a sorozatot, például úgy, hogy hiába nagyobbak a csatornák, mindig csak egy üzenetet rakunk bele. Tehát ha a szűkítési feltételekkel is találunk tesztet, akkor azt ugyanúgy felhasználhatjuk, és sikerült jelentős erőforrásokat megtakarítanunk.

Az előzetes mérések alapján kétféle mód tűnt a gyakorlatban is használhatónak:

- Szélességi bejárás és HC4 tömörítés: ez sajnos nem teljesen váltotta be a hozzá fűzött reményeket, az átmenetek jelentős részével nem birkózott meg. Körülbelül 190 lépést képes bejárni 1 GB memóriával a protokoll modelljén. Erre a modellre csak úgy lehetne használni, ha az állapotvektor nagyságát csökkentjük, például ha csak az épp aktuális formulához szükséges állapot és átmenet jelző biteket deklarálnánk globálisra, a többit lokális változóként vennénk fel.
- Bitstate mód és mélységi bejárás mélységi korláttal. A –wN paraméter helyes megadása kulcsfontosságú ilyenkor, [Holz03] szerint akkor jó, ha 2 ^ N = az ellenőrzésre szánt memória. Azonban ha túl nagyra választjuk, akkor jelentősen lassítja az ellenőrzést, ha túl kicsire, akkor pedig nem találunk bizonyos formulákhoz tesztet, azért mert az ütközések miatt az állapottér egy részét nem járja be.

Ilyen beállításokat alkalmazva is még több, mint két és fél óra volt a verifikáció, ezért a *tesztkészlet összeállítását* végző részt módosítottam a programban. Egy generált teszteset természetesen nem csak azt a tesztelési követelményt teljesíti, ami a hozzá tartozó formulában meg volt fogalmazva. A teszteset elkészítése után megvizsgálom, hogy milyen egyéb követelményeket fed le (például milyen állapotokat, átmeneteket érintett), és ezekhez akkor már nem kell tesztet generálni. Ehhez az ellenőrző lefutása után a modell ellenőrző által nyújtott ellenpélda leírásából egy részletesebb leírást készíttetek, ami tartalmazza minden egyes lépéshez az összes változó értékét.

13. táblázat: Tesztgenerálás a szinkronizációs protokollhoz

Paraméter	*Futási idő*[2]	*Előzőleg lefedett*	*Nem talált tesztet*
–m1000 –w31 (további fedés vizsgálat nélkül)	215m20.021s	0 %	0 db
–m1000 –w31	65m4.320s	65 %	0 db
–m1000 –w26	46m2.303s	62 %	2 db
–w26	56m47.658s	55 %	4 db
–w24	28m3.350s	48 %	8 db

Ennek az egyszerű heurisztikának az alkalmazásával is meglepően jó eredményeket érthetünk el. A futási idő fele részére csökkent, és a teszt követelmények 65 százalékához nem kellett futást generálni, mert már egy előző lefedte azt.
A táblázat többi sora pedig alátámasztja az eddig elmondottakat:

- Mélységkorlát megadásával csökkenthetjük a futási időt.
- A –w paraméter nagyobb értékeivel a futás lassabb.
- Ugyanakkor túl kicsi –w mellett nem biztos, hogy mindenre találunk tesztet.

A szakasz bemutatta, hogy a korábban vizsgált módszerek és eredmények nem lesznek elégségesek egy valós alkalmazás során. Azonban a SPIN bitstate üzemmódjával, a rendszer működésének szűkítésével és a már lefedett követelmények felhasználásával az automatikus tesztgenerálás módszere használhatóvá válik a gyakorlatban is.

[2] Ezeket a méréseket egy Pentium4 2.4GHz-es gépen, 1 GB memóriával végeztem.

5.4 További tesztkészlet optimalizációk

A szinkronizációs protokoll példán keresztül láthattuk, hogy valós alkalmazás esetén egy-egy teszt generálása igen erőforrás-igényes lehet. A szakasz végében bemutatott módszer segítségével azonban csökkenteni lehet azoknak a követelményeknek a számát, amikhez ténylegesen teszteket kell generálni. A módszert tovább lehet csiszolni. Ha például olyan állapotok vagy átmenetek lefedésével próbálkozunk, amik „mélyen" vannak, akkor valószínűleg azoknak a tesztjei több követelményt fednek le, tehát érdemes ezeket előre venni. Így megtakaríthatjuk a rövidebb tesztek generálását, amit később egy hosszabb úgyis lefed.

Tovább lehet szűrni a kapott tesztkészletet, ha megvizsgáljuk, hogy nincs-e olyan teszteset, ami egy másiknak a prefixe, ilyenkor a rövidebbik kihagyható.

Egy másik irányú megközelítés, ha az állapottér méretét *alkalmazásfüggő temporális logikai kifejezésekkel* „levágjuk". Jól alkalmazható ez, ha a modell egy nagyobb részétől nem függ az éppen tesztelendő rész, akkor egy megfelelő temporális logikai formula megadásával megakadályozhatjuk, hogy a modell ellenőrző feleslegesen bejárja azt a részt. A mobiltelefonos állapottérképen például a képernyőn lévő szöveg szerkesztését nem befolyásolja, hogy a kapcsolat milyen állapotban van.

6 Tesztkészlet illesztése implementációhoz

Az előző fejezetek bemutatták, hogy egy állapottérkép modellhez hogyan lehet hatékonyan –a modell nagy részét lefedő– teszteseteket generálni. Egy valós szoftverfejlesztés során ez azonban csak a tesztelés első lépése, a végcél az, hogy ezeket a teszteket végre tudjuk hajtani egy, a modell alapján elkészített implementáción. A modell alapú tesztgenerálás használhatóságát pedig végső soron ott tudjuk lemérni, hogy az implementációhoz milyen minőségű, például milyen kódfedést elérő teszteket tudunk készíteni. Jelen szakasz ennek a feladatnak a részlépéseivel foglalkozik, és az ötödik fejezetben ismertetett mobil mintapéldán keresztül bemutatja, hogy az absztrakt, modellen értelmezett tesztesetből hogyan származtathatunk az implementáción elvégezhető teszteket.

6.1 Állapottérképek implementálása

A teszteléshez először is szükségünk van konkrét implementációkra. Állapottérkép alapú modellekből a kód származtatására több elterjedt módszer szerepel az irodalomban. [PM03] áttekinti a legfontosabbakat, ezek közül az úgynevezett *nested switch* módszer alapján és egy már létező *kódgenerátor* program segítségével elkészítettem a mobil mintapéldát megvalósító Java nyelvű kódot.

6.1.1 Nested switch módszer

Viszonylag egyszerű, intuitív módszer. Két egymásba ágyazott switch szerkezet segítségével találja meg, hogy egy adott esemény hatására milyen állapotváltást és milyen akciókat kell végrehajtani. Egy-egy enumerációban tároljuk az állapotokat és a lehetséges bejövő eseményeket. A külső switch-et a jelenlegi állapot alapján ágaztatjuk el, azon belül pedig a bejövő esemény szerint. Az olyan eseményekhez, amikhez az adott állapotban van olyan átmenet, aminek ez a trigger eseménye, a case részben elvégezzük a szükséges műveleteket. A default ágban esetlegesen jelezhetjük, hogy az állapottérkép eldobta a kapott eseményt, nem volt hozzá engedélyezett átmenet. A 23. ábrán egy rövid kódrészlet szemlélteti a módszer működését.

```
public void handleEvent(EventType event)
{
    // <editor-fold desc="-----nested switch for the top automaton----">
    switch ( state[0] )
    {
        case PowerOff :
            switch ( event )
            {
                case evKeyNoHold :
                    state[0] = MobileState.PowerOn;
                    state[1] = MobileState.StandBy;
                    state[2] = MobileState.LineOK;
                    performAction( EventType.OpenDisplay );
                    break;

                default:
                    // do nothing
                    break;
            }
            break;
```

23. ábra: Nested switch kódrészlet

A nested switch módszer legnagyobb hibája, egyszerűségéből következően, hogy alapesetben nem kezeli a konkurens régiókat, ugyanis egy állapotot tároló változó van, ami szerint a külső switch elágazik. A mobil példában azonban szerepel ÉS jellegű finomítás is, így az alap módszert kicsit át kellett alakítani. Szerencsére nem voltak hierarchia szintek között átívelő átmenetek és más nehezen kezelhető konstrukciók, így azzal a módszerrel el lehetett készíteni a mobil állapottérképet megvalósító kódot, hogy az állapotot nem csak egy State típusú változóban, hanem egy State típusokat tartalmazó tömbben tárolta. Minden összetett állapotnak megfelelt egy változó, és a külső, a legfelső szintű állapotnak megfelelő switch `case PowerOn :` ágába további két switch került az *rCalling* és *rLineControl* két konkurens régiónak megfelelően.

Később grafikus felület és direkt Java interfész hívás alapú teszteléshez is fel lettek használva az elkészült példák. Ezért a működést megvalósító MobileNestedSwitch osztály mellé készült egy Java Swing alapú grafikus felület. A két réteg közötti kommunikáció Java Event-ek segítségével valósul meg, a grafikus felület implementál egy EventListener interfészt, és feliratkozik a MobileNestedSwitch osztály által küldött eseményekre.

6.1.2 Kódgenerátor alkalmazása

A nested switch alapú módszer bonyolultabb állapottérképekre már nem igazán alkalmazható, és az így elkészült kódnak a karbantartása is nehézkes. Ahhoz, hogy a kódot automatikusan lehessen származtatni az állapottérkép modellből már bonyolultabb szerkezetű kód szükséges. Az általam használt kódgenerátor, melyet a Méréstechnika és Információs Rendszerek Tanszék doktorandusz hallgatója, Pintér Gergely készített [PM03], a harmadik fejezetben ismertetett EHA formalizmust használja fel. A program tartalmaz általános osztályokat, melyek az egyes konstrukciók (automata, átmenet, stb.) működését definiálják, és a konkrét modell elemeit ezekből az osztályokból származtatja.

A generált kód három csomagot tartalmaz:

- *dynamic_behaviour*: az absztrakt alaposztályokat tartalmazó csomag. Az Automaton, Configuration, Event, State és Transition osztélyok írják le, hogy milyen adatokat tárolunk az állapottérképről. Az EHA osztály metódusai pedig a működési szemantikát tartalmazzák, pl. a collectEnabled az engedélyezett átmenetek kigyűjtését végzi a dispatchEvent az eseménykiválasztást.
- *generated_events*: minden egyes eseményből készül egy osztály, mely az Event-ből származik. Az esemény egyedi azonosítóját, az XMI leírásban használt azonosítót és a szöveges leírását tartalmazza.
- *generated0*: minden egyes állapothoz és átmenethez tartalmaz egy generált osztályt. Az alap adatokon (név, ID) kívül a be- és kilépési akciókat, valamint a tüzeléskor bekövetkező akciókat lehet itt külön definiálni. Ezen kívül készül egy EHA0 osztály is, mely összefogja a többi a hierarchia definiálásával.

A mobil állapottérképből 54 osztályt generált 60 KB-nyi Java forrással. Az automatikus generálásnak köszönhetően ez már egy sokkal általánosabb kód, az állapottérképekben megtalálható konstrukciók nagyobb részét fedi le, mint a kézzel elkészített nested switch alapú megoldás, így az ezzel végzett mérések jobban tükrözik a generált tesztek általános használhatóságát.

6.2 Konkrét tesztesetek származtatása

Az elkészült implementációk eltérő struktúrájú kóddal rendelkeznek, különböző felület biztosítanak az események feldolgozására, az eseményekre és akciókra belső azonosítókkal hivatkoznak, mely különbözik az állapottérképen definiált nevektől. Így ahhoz, hogy a generált eseménysorozatokat tartalmazó teszteket végre tudjuk hajtani, teszt interfészeket kell készíteni, melyek meghajtják az elkészült implementációkat.

Egy konkrét teszteset a következő részeket kell, hogy tartalmazza:

- „Setup" kód: előállítja a teszteset kezdőállapotát, létrehozza a szükséges objektumokat, struktúrákat, amik a modellben definiált kezdőállapotot reprezentálják.
- Események küldése és akció ellenőrzése: minden egyes, az absztrakt tesztesetben szereplő eseményküldéshez generálni kell egy kódrészletet, mely feldolgoztatja az eseményt a fogadóval, és ellenőrzi, hogy a tesztesetben szereplő akciónak megfelelő műveletek hajtódtak-e végre. Ehhez definiálni kell, hogy az események és akciók minek felelnek meg az implementációban (pl. egy signal küldése, változó értékének állítása, stb.).
- „TearDown" kód: a teszt végén lévő takarító kód, felszabadítja a lefoglalt erőforrásokat.

A következő szakaszban láthatjuk, hogy fenti lépéseknek megfelelő kód előállításának egy részét hogyan lehet automatizálni.

6.3 Teszt futtató keretrendszerek

Az elkészült konkrét tesztesetek végrehajtását és kiértékelést végezhetjük kézzel is, de sokkal hatékonyabb valamelyik elterjedt teszt futtató keretrendszer használata. Az ilyen rendszerek a tesztelendő rendszer és a tesztkészlet definiálása után a teszteket elvégzik és kiértékelik, majd az eredményt eltárolják. Így később visszakereshető, kielemezhető, hogy az implementáció különböző verziói hogyan szerepeltek. Az ilyen keretrendszereknek két jól elkülöníthető fajtája van, az egyik, mely alacsony, modul szintű, úgynevezett unit teszteket futtat, a másik pedig az elkészült program felhasználói felületéről indítja a teszteket. Munkám során megvizsgáltam egyet-egyet a két típusból, és példákon keresztül felmértem, hogy hogyan illeszthetőek a tesztgenerátor által készített tesztesetek ezekhez a rendszerekhez.

6.3.1 JUnit

A unit tesztek készítésének módszere az Extreme Programming [Beck04] módszertan ajánlásaiban jelent meg, és gyorsan elterjedt a szoftverfejlesztők körében. A programozó, még a tényleges kód elkészítése előtt, az osztályához teszteket ír, mely az elvárt funkcionalitást ellenőrzi. Ezután pedig egy folyamatos kód írás – teszt futtatás – hiba javítás ciklus következik, így a program helyességéről nagyon hamar kap a fejlesztő visszajelzést. A folyamatos teszteléshez viszont elengedhetetlen a unit tesztek végrehajtásának automatizálása, ezért születtek meg a különböző XUnit keretrendszerek, ilyen például a Java nyelvhez íródott JUnit [JUnit]. Egy JUnitban futtatható teszt egy Java osztály, mely a következő tulajdonságokkal rendelkezik:

- A TestCase osztályból származik, ezzel jelzi a keretrendszernek, hogy őt futtatni kell.

- Felüldefiniálhatja a setUp() és tearDown() metódusokat, ezek az egyes tesztek kezdőállapotának beállítására és utána a takarításra szolgálnak.
- Minden metódusa, amely publikus és test-tel kezdődik egy teszteset lesz.
- A TestCase osztály assert kezdetű metódusaival tudja definiálni az elvárt működést, például az assertEqual azt ellenőrzi, hogy a két paramétere, az elvárt és a teszt során kapott objektum, egyenlő-e.
- A teszt metódusokat tesztkészletekbe (suite) gyűjtheti, és így lehetőség van arra, hogy a teszt osztályban definiált tesztek egy részét futtassuk csak le egyszerre.
- A tesztek futtatására grafikus és parancssori felületet is biztosít a keretrendszer, melyen megjelenik, hogy a futtatott tesztek közül melyiknek mi lett az eredménye.

A JUnit tehát egy könnyen használható, rugalmas keretrendszer unit tesztek futtatásához.

Ha a generált absztrakt teszteseteket megfeleltetjük a 8.2 részben leírt módon JUnit teszteknek, akkor a kapott teszt kód nagy része ismétlődik, az egyes események küldése ugyanúgy zajlik minden teszt metódusban. Így néhány előre definiált sablon segítségével az absztrakt tesztesetből generálható a JUnit-os Java kód. Elkészítettem egy szkriptet, mely ezt a feladatot elvégzi.
A szkript a következő sablonokat használja:

- Teszt fájl eleje, vége,
- Teszt metódus eleje, vége,
- Esemény küldése, akció ellenőrzése,
- Esemény és akció nevek megfeleltetése a kódban használt neveknek.

Az utolsó kettő az, ami a tesztelendő rendszerenként különbözik, és a tesztelőnek kézzel kell elkészítenie minden egyes alkalommal.

A generálás menete:

1. A bemenő paraméter az elkészítendő teszt osztály neve. Ezt behelyettesítve elkészíti a teszt fájl elejét, mely a JUnit specifikus inicializáló kódokat tartalmazza.
2. A megadott könyvtárban lévő absztrakt tesztesetek mindegyikéhez generál egy teszt metódust. Egy XSLT transzformáció segítségével kinyeri sorrendben az esemény és akció neveket. Minden egyes eseményt –a megfeleltetés segítségével– leképezi az implementációban használt nevére, majd behelyettesíti az esemény küldő sablonba. A helyes választ pedig az akció ellenőrző sablon behelyettesített változatával ellenőrzi (ami általában valami assert konstrukciót tartalmaz).
3. Ha minden absztrakt tesztesetet feldolgozott, akkor zárásként generálja a teszt fájl végét.
4. A generált fájlban jelölve vannak azok a részek, amik a konkrét tesztelendő rendszertől függően mások és mások, és még a tesztelőnek kell kitölteni, pl. a setUp kód.

A 24. ábrán látható példa szemlélteti, hogy milyen kód keletkezik a generálás után. A függelék C. pontjában megtalálható a teljes kód és a felhasznált sablonok egy része is.

```
public class MobileNestedSwitchStateTest extends TestCase implements ActionEventListener
{
    // # ------- Private fields -------
    private MobileNestedSwitch mobile = null;
    private int actionReceived = -1;
    // # ------- Private fields -------

    // # ------- SUT specific methods -------
    //TODO
    // # ------- SUT specific methods -------

    /** Test for mobile.CallWait state */
    public void testInMobileCallWait()
    {
        System.out.println("InMobileCallWait test");
        mobile.handleEvent( EventType.stringToInt( "evKeyNoHold" ) );
        System.out.println( "Action received: " + EventType.intToString( actionReceived ) );
        assertEquals( EventType.OpenDisplay, actionReceived );
    }
```

24. ábra: generált JUnit teszt

A szkript és az alap sablonok elkészítése után mindkét implementációhoz generáltam teljes állapotfedést és átmenet fedést megvalósító JUnit tesztkészletet. A 14. táblázatban látható, hogy minimális saját kód és a tesztgenerátor által elkészített absztrakt tesztesetek segítségével hosszú teszt kódok is könnyedén származtathatóak.

14. táblázat: JUnit tesztek generálása, az implementációs specifikus kód sorainak száma és az ennek a segítségével generált tesztfájl hossza.

	Implementáció specifikus kód	*Tesztfájl sorainak száma*
Nested switch	9	470
Kódgenerátor	11	515

6.3.2 Rational Testmanager és Robot

A Rational TestManager [TestManager] a Rational tesztelési termékcsaládjának központi komponense, a teljes tesztelési folyamat támogatását végzi. A felhasználói követelményekből tesztelési célokat lehet definiálni, majd ezeket konkrét tesztkészleteknek megfeleltetni, ezzel ellenőrizve, hogy minden követelményt megfelelően tesztelünk-e. Egy központi helyen tárolja az összes tesztesetet, és a hozzájuk tartozó meta információt (pl. milyen konfiguráción kell lefuttatni), ezzel könnyítve a tesztek menedzselését. A tesztek végrehajtását automatizálja, az egyes tesztesetekből komplex futtatási sorozatokat definiálhatunk. A különböző lefutások eredményeit később analizálhatjuk, meg lehet például határozni, hogy mekkora kódfedést érnek el a tesztek, a legutóbbi verzió óta mennyi, régebben sikertelen teszt volt most eredményes. A program manuális teszteket, grafikus felhasználói felület, webes és Java alapú programok funkcionális tesztjét is támogatja.

A Rational Robot [Robot] nevű eszközzel pedig a Testmanagerben definiált teszteset implementálását és futtatását végezhetjük el. A Robot a felhasználói felület tesztelő alkalmazások közé tartozik. Működésének alapja, hogy rögzíti a tesztelő felhasználói felületen végzett tevékenységét, majd ebből egy SQABasic (a Robot saját nyelve) szkriptet generál, mely később visszajátszható. Az SQABasic a VBScript nyelv kibővítése a tesztelés során gyakran használt funkciókat megvalósító konstrukciókkal, pl. ellenőrző rutinok, tesztadatok sorozatát tartalmazó tárolók elérése, hibák naplózása. A grafikus felület elemeit képes felismerni, így az egyes vezérlőkre azok forráskódban

definiált nevével hivatkozhatunk, nem pedig a pozíciójuk koordinátáival, mely pl. egy más képernyőfelbontásban más lehet.

A mobil példák illesztéséhez az alkalmazást kibővítettem egy egyszerű grafikus felülettel. A bejövő esemény nevét megadjuk egy szövegdobozban, egy gomb megnyomása után a program feldolgozza az eseményt, és kiírja a válaszként adott akció nevét. A helyes működést úgy lehet ellenőrizni, hogy megvizsgáljuk, hogy a kapott akció megegyezik-e a várt akcióval. A következő kód szemlélteti, hogy hogyan lehet ezt SQABasic nyelven megfogalmazni.

```
1       Sub Main
2           Dim Result As Integer

3           'Initially Recorded: 2005.04.07.  14:48:55
4           'Script Name: mobileNestedSwitchSpeak
5           StartJavaApplication  Class:=
                "testGenerator.MobileNestedSwitchGui",
                Working:="G:\classes", JvmKey:="Java"

6           Window SetContext, "Class=testGenerator", ""

7           InputKeys "evKeyNoHold"
8           PushButton Click, "JavaClass=MobileNestedSwitchGui;\
                ;Type=PushButton;Name=Handle"

9           Result = LabelVP (CompareProperties,
                "JavaClass=MobileNestedSwitchGui;\;
                Type=Label;Name=OpenDisplay", "VP=OpenDisplay")

10          Window CloseWin, "", ""

        End Sub
```

A szkript az 5. sorban elindít egy Java alkalmazást, a 7. sorban begépeli az alapértelmezés szerint kiválasztott beviteli mezőbe az evKeyNoHold szöveget, majd megnyomja a Handle nevű gombot. Az eredmény ellenőrzését a LabelVP függvényhívással, egy úgynevezett *verification point*, segítségével végzi el, ahol a címke vezérlő tartalmát az OpenDisplay konstanssal hasonlítja össze.

Könnyen látható, hogy a 8.2-ben szereplő általános minta, inicializáló kód – esemény küldése – akció ellenőrzése – záró kód, itt is megfigyelhető. A JUnit tesztek generálásához felhasznált szkriptben csak a sablonok tartalmát kellett megváltoztatni, és azután a Rational Robothoz is tudta illeszteni az absztrakt teszteseteket.

6.4 Kódfedés mérése

Az elkészült tesztek egyik legalapvetőbb mérőszáma, hogy mekkora fedettséget biztosítanak a forráskódon. A száz százalékos fedés ugyan még nem bizonyítja, hogy a tesztjeink tényleg jók, és minden, a követelmények szempontjából fontos esetet megvizsgálnak, de legalább azt biztosan tudjuk, hogy minden kódsort, valamilyen körülmények között legalább egyszer lefuttattunk a tesztelés során.

6.4.1 Kódfedési metrikák

Kódfedés mérésére többféle metrikát alkalmaznak. A legegyszerűbb az *utasításlefedettség* (statement coverage), azaz, hogy a kódban szereplő utasítások hány százalékát hajtottuk legalább egyszer végre (szoktak kódsor lefedettséggel is hivatkozni

erre a mérőszámra, de ez a kifejezés nem pontos azon nyelvek esetén, ahol egy sorban több utasítás is lehet). Ennél bonyolultabb mérőszám a *döntési ág fedettség* (branch coverage). Ilyenkor azt mérjük, hogy a kódban szereplő feltételek lehetséges kimenetei közül hányat fedtünk le. Száz százalékos döntési ág lefedettség esetén is kimaradhatnak fontos részek. A következő kód estén például a = 2 és a = 0, b = true teljes döntési ág lefedettséget biztosít, viszont a lusta kiértékelés miatt c.subString hívás nem hívódik meg, így nem találja meg azt a hibát a tesztkészlet, ha előtte c-nek nem adtunk megfelelően értéket.

```
if ( (a < 1) && ( b || (c.subString(0,2) == „be” ) ) )
```

Az ilyen esetek elkerülése végett definiálták a bonyolultabb fedési mérőszámokat. Az *út lefedettség* (path coverage) esetén a feltétel az, hogy a program forrásából konstruált *folyamat vezérlési gráfon* (Control Flow Graph) hány utat járunk be a lehetségesek közül. A magas út lefedettség a program kimerítő tesztelését biztosítja, azonban bonyolultsága miatt kevés eszköz támogatja a mérését.

A fedést mérő programok alapvető működési elve az *instrumentálás*. A forráskódot vagy a lefordított binárist (Java kód esetén a byte kódot) kiegészítik olyan jelző utasításokkal, amikkel számon tudják tartani, hogy az utána következő utasítás meghívódott, az adott feltétel mire értékelődött ki.

6.4.2 Kódfedést mérő programok

A mobil mintapélda kapcsán két programot vizsgáltam meg. A Rational teszt programcsomag része a *PureCoverage*, mellyel C/C++, Java és .NET-es menedzselt kódú programokon lehet utasítás és metódus lefedettséget mérni. Vagy a grafikus felületén megadjuk neki a futtatandó programot, és, ha vannak, a parancssori argumentumait, majd a program befejezéséig gyűjti a futás közben a fedési mérőszámokat. Vagy pedig a TestManagerben definiált és futtatott tesztekhez állítható be, hogy a teszt eredményének ellenőrzése mellett a futási mérőszámokat is gyűjtse. A PureCoverage használata egyszerű, azonban csak a legalapvetőbb mérőszámot ismeri.

A másik program, amit használtam a Cenqua cég *Clover* [Clover] nevű terméke. Többféle verziója létezik, modulként csatlakoztatható az ismertebb grafikus fejlesztői keretrendszerekhez (Eclipse, NetBeans, JDeveloper), másik változata pedig a különböző build rendszerekbe illeszthető be. Az utóbbi fajtát használtam, azon belül is az *Apache Anthoz* készített típusát. Az Ant a C-ből ismert make fájlok kiváltására íródott, és Java környezetben a legelterjedtebb build rendszer. Egy XML formátumú fájl segítségével lehet definiálni, hogy a binárisok előállításához milyen lépéseket kell megtenni, milyen fájlokból kell kiindulni. Rengeteg beépített műveletet, úgynevezett taskot, tartalmaz, pl. fájl műveletek, Java fordító meghívása, levél küldés, ezen kívül pedig a keretrendszer nyílt, tovább bővíthető. A Clover is ezt használja fel, két saját műveletet definiál. A clover-setup létrehoz egy bináris állományt, melyben a mérési eredményeket tárolja, és elvégzi a forrásfájlok instrumentálását. A clover-report pedig az összegyűjtött adatokból jelentést készít html vagy xml formátumban. A 25. ábra szemlélteti az Ant leíró fájljának felépítését, egy teljes példa pedig megtalálható a függelékben.

```
<project name="clover.generated" default="main" basedir=".">
    <property name="build.dir" value="${basedir}\build" />
    <taskdef resource="clovertasks" />
    <target name="main" depends="clean,init,with.clover,compile,test,clover.html" />
    <target name="with.clover">
        <clover-setup initString="mycoverage.db" />
    </target>
    <target name="compile">
        <javac srcdir="${src.dir}" destdir="${build.classes.dir}" />
    </target>
```

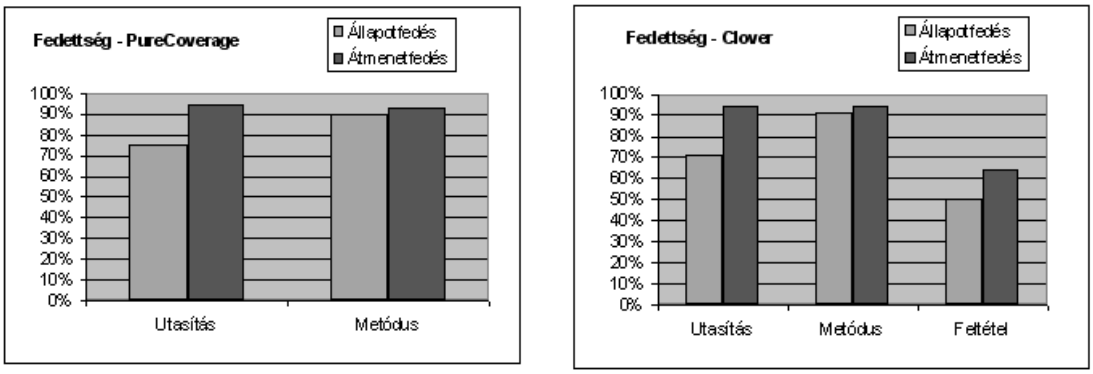
25. ábra: build.xml részlet

A Clover utasítás, metódus és döntési ág lefedettséget képes mérni, és az Ant integrációnak köszönhetően aprólékosan testre szabható, hogy a méréseket milyen fájlokra és a build és teszt folyamat melyik fázisaiban végezzük el. Az elkészült jelentésekben az összesített információk mellett akár azt is megnézhetjük, hogy egy-egy adott sor hányszor hajtódott végre a futás során. A használat során egyetlen problémám volt vele, hogy még nem kezelte az 1.5-ös Javaban megjelent nyelvi újításokat, például az enum kulcsszót, így némileg át kellett írnom a nested switch módszerű példakódot, hogy el tudja végezni az instrumentálást.

A fenti két programon kívül kipróbáltam még a JCoverage-t is. Ez egy ingyenes, utasítás és döntési ág lefedettséget mérő program, mely, a Cloverhez hasonlóan, az Ant build fájlokba képes beépülni. A Java byte kódon végzi el az imstrumentálást, azonban az 1.5-ös Javat ez sem támogatja még. Így végül a mobil példához kapcsolódó méréseket a Cloverrel végeztem, annak ugyanis több információt tartalmaznak az elkészült jelentései, például számol metódus lefedettséget is.

6.4.3 Mérési eredmények

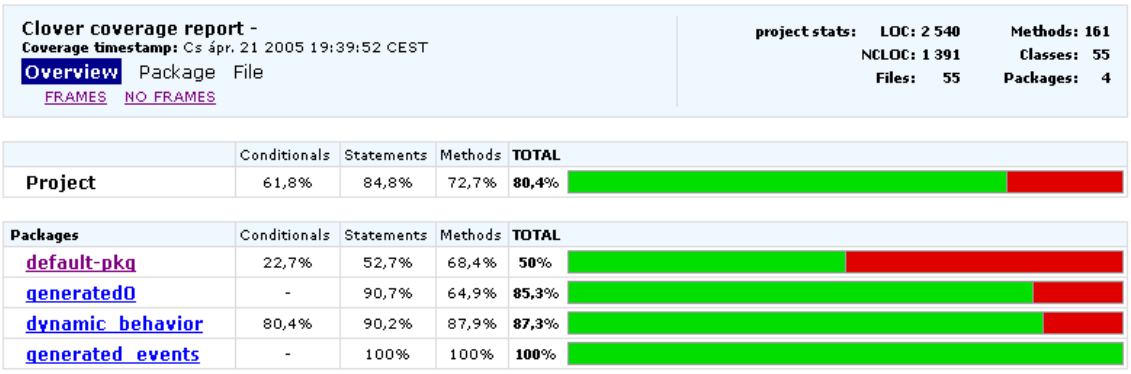
A kód lefedettséget a JUnit tesztek segítségével mértem. A PureCoverage esetén a JUnit szöveges tesztfuttató programját indítottam el, paraméterként átadva neki a generált teszt fájlt. A Clover esetén az Ant build.xml-ben definiálható JUnit-os tesztelési lépés is, így ott egyszerűen csak a build folyamatot kellett elindítani. A teljes állapot és átmenet lefedést biztosító tesztkészletek esetén a következő eredményeket kaptam.



26. ábra: Fedési eredmények – Nested switch

A két eszközzel nagyjából ugyanazokat az eredményeket kaptam. Állapot lefedés esetén az utasítások kb. 70%-t fedték le a tesztek, míg átmenet fedés estén 94%-os lefedettséget sikerült elérni, ami átlagos, nem biztonságkritikus szoftverek esetén egész jó kiinduló tesztkészlet. Az utasítások maradék öt százaléka főleg hibakezelő kód, nem

megfelelő eseménynév átadása az állapottérképnek, adott állapotban nem értelmezett esemény kezelését végző utasítások, stb. Ezek nyilvánvalóan nem hajtódnak végre, hisz csak pozitív teszteket generáltunk. Ha a tesztek között szerepeltek volna a 3.2.1 részben leírt implicit átmeneteket lefedő tesztek, akkor ezeket az utasításokat is végre hajtottuk volna. A feltétel lefedettség nagyon alacsonynak tűnhet, ennek oka, hogy a példában nagyon kevés (3 if és 2 switch) feltétel szerepelt, így néhány eset kihagyása is drasztikusan csökkentette a végeredményt.



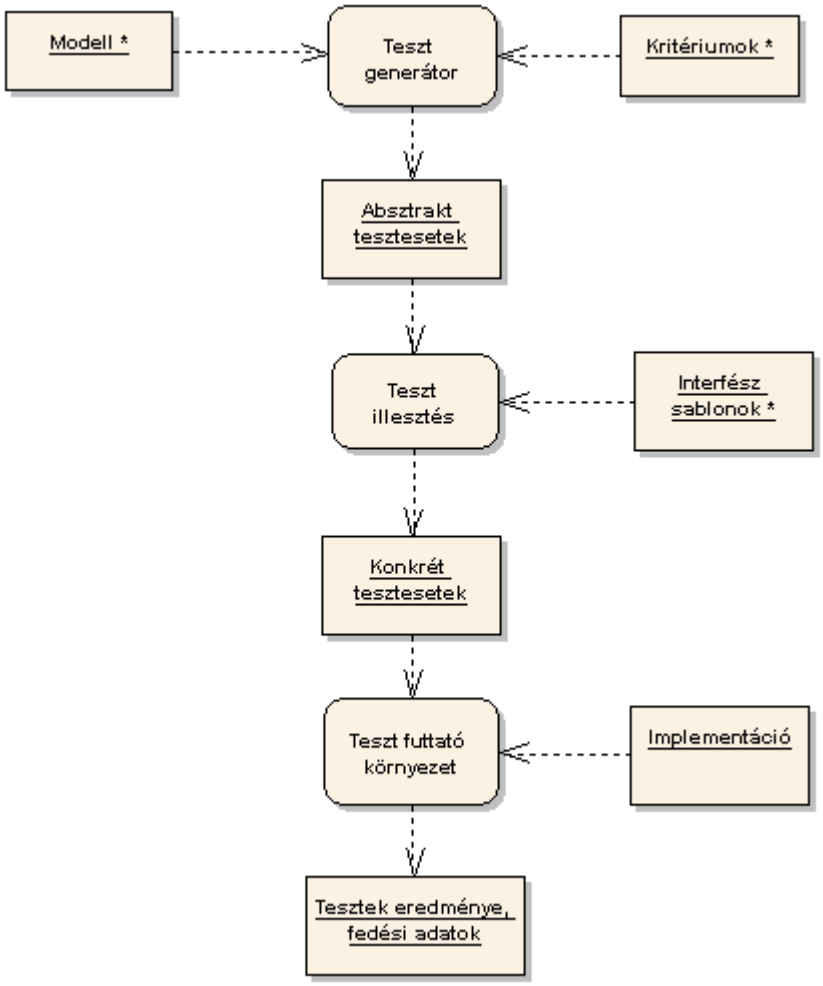
27. ábra: Fedési eredmények – kódgenerátor

A 27. ábrán a Clover által generált HTML jelentés egy részlete látható, mely jól összefoglalja, hogy a tesztkészletek milyen fedést értek el a kódgenerátorral kapott implementáción. A PureCoverage használata esetén kapott eredményektől való eltérés fél százalékon belül volt, ezért szerepel itt az összehasonlító diagram helyett inkább egy kicsit részletesebb jelentés. A default-pkg csomagban található osztályok jelentős része a parancssorról való beolvasás és kiírás feladatait végezik el, ezért nem hívódtak meg a JUnit-os tesztek során, ennek köszönhető a csak 50%-os lefedettség.

6.5 Tesztelési folyamat összefoglalása

A fejezet zárásaként az 28. ábra összefoglalja, hogy mely lépéseket kellett megtennünk, míg a modellen megfogalmazott kritériumtól eljutottunk egy teszt lefuttatásához az implementáción. A csillaggal jelölt lépések azok, amiket részben kézzel kellett elvégezni. A mobil telefon mintapéldáján látott módon először elkészítjük az állapottérképes modellt. A tesztjeink később olyan részletesek lesznek, amilyen részletes ez a modell, tehát a modell absztrakciós szintje határozza meg, hogy később mi lesz egy elemi lépés a tesztekben. Személyes tapasztalatom az, hogy egy általános üzleti alkalmazás esetén nehéz állapottérképekkel megfogalmazni a működést, ott az adatszerkezeteket leíró osztálydiagramok, és az egyes hívási láncokat definiáló szekvencia vagy aktivitás diagramok dominálnak. Azonban egy beágyazott, reaktív rendszer vagy egy protokoll esetén az állapottérkép kellően részletes információt tud szolgáltatni.

Ha a modell elkészült, a megfelelő kritérium megadásával a teszt generátor program előállít egy, a kritériumot lefedő tesztkészletet. A generátor programban legfeljebb a tesztesetek maximális hossza és az állapottér becsült mérete beállításokat kell a modell méretéhez igazítani, a SPIN modell ellenőrző alapos ismeretét feltételező beállításokat nem kell módosítani.



28. ábra: Tesztelési folyamat

Az absztrakt tesztesetek leképezéséhez az implementációhoz való kapcsolódást leíró sablonokat kell az adott rendszerhez igazítani. A mobil példa kapcsán láthattuk, hogy egyszerűbb program esetén ez 10-20 sornyi kódot jelent, a tesztesetek kódjának maradék részét automatikusan lehet származtatni.

Ezután már csak végre kell hajtanunk a kapott tesztjeinket. Megfelelő teszt futtató keretrendszer esetén pedig nem csak a tesztek eredményét hanem azok kód lefedettségét is egyből megkaphatjuk.

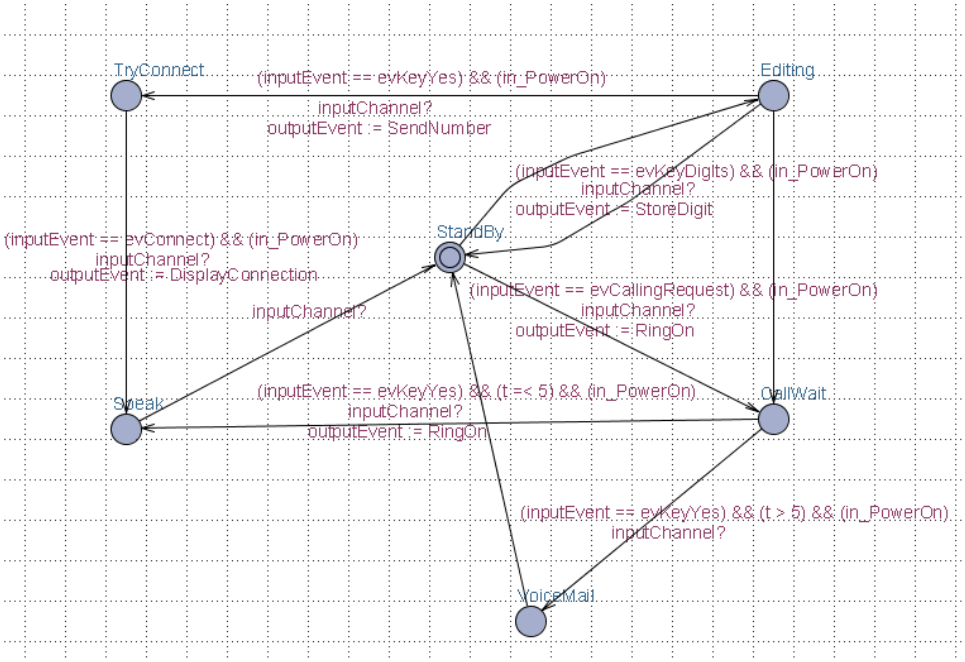
7 Kiterjesztés valósidejű rendszerekre

Beágyazott rendszerek esetén nagy jelentősége van a *valósidejű rendszereknek*, ahol nem csak nyújtani kell valamilyen szolgáltatást, de azt egy előre megadott határidő lejárta előtt kell teljesíteni. Ezeknek a tervezése és modellezése más metodikát és eszközöket kíván. Az ilyen rendszerek ellenőrzéséhez is speciális modell ellenőrzők készülnek.

7.1 Tesztgenerálás az Uppaal modell ellenőrzővel

Valósidejű rendszerek ellenőrzésére használható például az *UPPAAL* nevű eszköz [UPPAAL]. Az idő kezeléséhez speciális, óra típusú változókat használ, és ezeknek az értékeire lehet hivatkozni az őrfeltételekben és a verifikációban. Az ellenőrzésben CTL-hez nagyon hasonló, csak szűkebb, a formulák konstruálását jobb megkötő szabályú kifejezéseket lehet megadni, de a 3.4-ben szereplő tesztkövetelmények azért még itt is könnyedén felhasználhatóak.

Lássunk egy példát az Uppaal használatara!



29. ábra: Mobil példa részlete az Uppaalban

Az Uppaal alapeleme a processz, ami tulajdonképpen egy automata. A mobil példát négy processzel lehet megvalósítani: a bemeneti eseményeket generáló, a felső szintű és a két konkurens régiót megvalósító automatával (ebből az egyik konkurens régióé látható a képen). Bevezettem egy VoiceMail állapotot, hogy be tudjam mutatni az időzítési kritériumok alkalmazását. A CallWaitből kivezető két átmenet őrfeltételét megváltoztattam, hogy figyeljék a t idő típusú változót. Ha a hívás utáni öt időegységen belül nem veszik fel a telefont, akkor a hangposta jelentkezik be. Az őrfeltételben látható in_PowerOn feltételre a szülő automatával való összehangolás miatt van szükség, ez az automata csak akkor léphet, ha a legfelső szintű a PowerOn állapotban

van (az Uppaal nem támogatja a hierarchikus finomítást, úgyhogy kénytelen voltam ehhez a megoldáshoz folyamodni).

Az `A[](! mobile2.VoiceMail)` formulával teszteltem a működését. Az Uppaal nemcsak grafikus, hanem konzolos felületet is biztosít a verifikációhoz. Az ellenőrzés során használhatunk szélességi és mélységi bejárást is, itt is van állapottömörítő és *bitstate hashing* funkció. Ezen kívül képes arra, hogy az állapotteret újrahasznosítsa több formula ellenőrzése során, vagy pedig a legrövidebb vagy a leggyorsabb futásokat keresse meg. A kiadódó trail fájlból is viszonylag egyszerűen ki lehet olvasni a szükséges információkat:

```
State:
( input.sendInput mobile.PowerOn mobile1.LineOK
mobile2.CallWait )
t=0 inputEvent=20 outputEvent=14 in_PowerOn=1 #depth=4

Transitions:
  input.sendInput->input.sending { 1, tau, inputEvent :=
evKeyYes }

State:
( input.sending mobile.PowerOn mobile1.LineOK mobile2.CallWait
)
t=0 inputEvent=28 outputEvent=14 in_PowerOn=1 #depth=5

Delay: 6
State:
( input.sending mobile.PowerOn mobile1.LineOK mobile2.CallWait
)
t=6 inputEvent=28 outputEvent=14 in_PowerOn=1 #depth=5

Transitions:
  input.sending->input.sendInput { 1, inputChannel!, 1 }
  mobile2.CallWait->mobile2.VoiceMail { inputEvent == evKeyYes
&& t > 5 && in_PowerOn, inputChannel?, 1 }

State:
( input.sendInput mobile.PowerOn mobile1.LineOK
mobile2.VoiceMail )
t=6 inputEvent=28 outputEvent=14 in_PowerOn=1 #depth=6
```

Látszik, hogy a CallWait állapotban van a rendszer, majd tüzel a sendInput átmenet, és az evKeyYes eseményt nyújtja. Ezután a Delay: 6 sorra hívnám fel a figyelmet, ez jelzi az idő múlását. Kész rendszer tesztelésekor ez adja meg a szükséges teszt időzítések értékét. Mivel öt időegység letelt, a CallWaitből VoiceMail átmenetbe átvivő átmenet is tüzelhet, és így eljutunk a végállapotba, ahol a mobile2.VoiceMail lesz az aktív állapot.

Az Uppaal modell ellenőrző tehát alkalmas lehet tesztgenerálási feladatokra, megvannak benne azok a funkciók, amik a Spinben és SMV-ben hasznos bizonyultak a folyamat során, ráadásul a modell alapján automatikusan képes előállítani a tesztek közötti időzítések értékét. Ez alkalmassá teszi valósidejű rendszerek validációs teszt sorozatainak generálására.

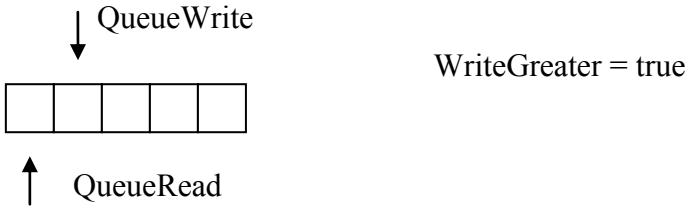
7.2 UML állapottérkép megvalósítása Uppaal modellben

A tesztgeneráló program átültetéséhez az Uppaal környezetébe, a legfontosabb feladat az UML állapottérkép → nem hierarchikus időzített automaták transzformáció megtervezése és helyes megvalósítása. A folyamatot itt is célszerű két lépésben megvalósítani, egy köztes formalizmus felhasználásával. [DMY01] a *hierarchikus időzített automatákat* (Hierarchical Timed Automaton) javasolja erre, és megad egy kisimítási algoritmust, mely az Uppaal által használt időzített automata dialektusra képezi le a HTA-kat. Egy pacemaker modellen szemléltetik a módszer működését, és elkészítettek egy Java nyelvű implementációt, mely elvégzi a két, XML dokumentumokban megadott, forma közötti transzformációt. A Vanilla-1 kódnevű program legutóbbi frissítése azonban 2001-ben történt, és úgy néz ki, hogy egyenlőre nem is tervezik a folytatását, így sajnos az nem használható fel.

A transzformáció bonyolultságának és a problémás feladatok feltárásának érdekében a mobil példát megvalósítottam az Uppaal modell ellenőrzővel.

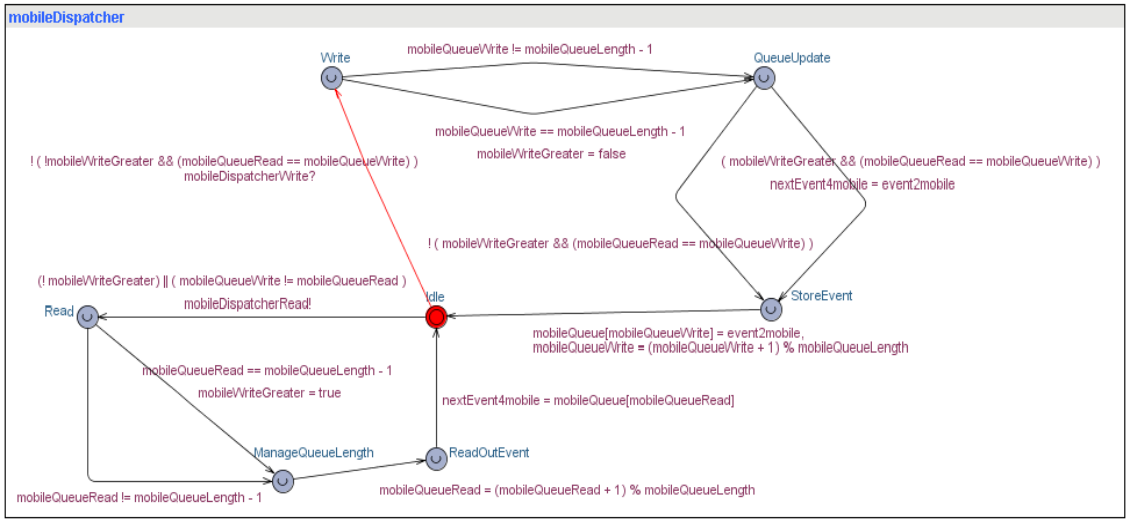
A hierarchikus állapottérképek megvalósításához minden egyes összetett, tovább finomított állapotot egy külön automata valósít meg. Az állapottérképen egy állapot nem lehet aktív, ha a szülő állapota nem aktív, így ezt korlátozni kellett az Uppaal modellben is. Mivel nem lehet procedurális utasításokat megadni, csak globális változókat, és az átmenetek őrfeltételét illetve akcióit, bool típusú globális változókkal jelöltem, hogy éppen aktív-e egy szülőállapot, és a gyermek automata minden átmenetének őrfeltételében szerepel ez a változó feltételként.

Az eseménykezelő megvalósítását egy külön processz végzi. Az Uppaalban csak szinkronizáció jellegű csatornákat lehet definiálni, azok adatok átadására nem alkalmasak, nem úgy, mint a SPIN-ben szereplő konstrukciók. Ezért a több elemet tároló, FIFO jellegű eseménysort egy tömbbel valósítottam meg. Két egész változó megfelelő karbantartásával egyszerűen képezhető egy körpuffer a tömbből.



30. ábra: Körpuffer megvalósítása

A 30. ábrán látható módon az egyik változó azt jelzi, hogy hova kell beírni a következő elemet, a másik pedig, hogy honnan kell majd olvasnunk. Ha a QueueWrite eléri a tömb határát, és léptetjük még egyet, hogy a nulla indexre mutasson, akkor WriteGreater jelzőbitet hamisra kell állítani. Olvasni akkor lehet, ha WriteGreater hamis (ugyanis ilyenkor a csatorna QueueRead index feletti részét biztos teleírtuk már új elemekkel) vagy pedig ha a WriteGreater igaz és QueueWrite nagyobb, mint QueueRead (azaz írtunk már be elemet, és továbbléptettük a mutatót).



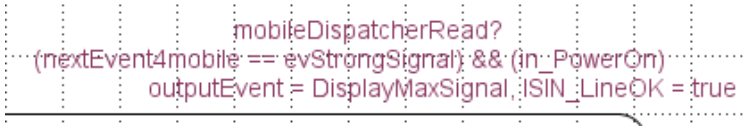
31. ábra: Dispatcher megvalósítása Uppaalban

A 31. ábra szemlélteti, hogy egy Uppaal automatával hogyan is lehet ezt a logikát megvalósítani. Az Idle állapotban (az ábra közepén) kezd az eseménykezelő, a jobb felső rész az eseménysorba beírást kezeli, tehát amikor eseményt küldenek az állapottérképnek, a bal alsó rész pedig az olvasást, tehát amikor kiolvas egy eseményt az állapottérkép. Két további változó könnyíti meg, hogy a többi processz átmeneteiből könnyen kezelhessük az eseménykezelőt és elfedjük a belső megvalósítását. Ezek az event2mobile, ezt állítja be az, aki küldeni akar egy eseményt az állapottérképnek, a nextEvent4mobile pedig azt tárolja, hogy mi lesz a következő esemény, amit az állapottérképnek fel kell dolgoznia. Ezek segítségével később az Uppaal által generált kimenetben is könnyebb lesz a számunkra érdekes részeket megtalálni.

Azt, hogy egyszerre csak egy folyamat írhasson az eseménykezelőbe, egy mobileDispatcherWrite nevű szinkronizációs csatorna biztosítja. Ha ez szabad, és hajlandó szinkronizálni (azaz az Idle állapotban van) és teljesül az írás feltétele, van szabad hely még az eseménykezelőben, akkor megkezdődhet az írás folyamata. A Write → QueueUpdate átmenetek, ha a QueueWrite átfordul, akkor hamisra állítják a WriteGreater bitet. Ha ez az első írás, akkor beállítjuk a nexEvent4mobile változót. Ezt később az olvasás rész teszi meg, de első íráskor az még nem futhatott le. Végül a StoreEvent → Idle átmenet elmenti a beírt eseményt, és továbblépteti a következő helyre a QueueWrite mutatót.

Az olvasás hasonló módon működik. Itt az indulási feltétel, hogy van már esemény, amit ki lehet olvasni. Első lépésként itt is állítjuk a WriteGreater változót, ha szükséges. Majd léptetjük a mutatót, és a nextEvent4mobile változóban eltároljuk, hogy mi a következő esemény. Azért van erre szükség, mert így tudjuk akkor majd a mobil processz őrfeltételeiben vizsgálni, hogy tüzelhető-e egy átmenet.

Ezek alapján egy átmenet a 32. ábrán látható módon néz ki a mobil processzben.



32. ábra: Átmenet az Uppaalban

Az átmenet tüzelésének feltétele, hogy tudjon a mobileDispatcherRead csatornával szinkronizálni, a trigger eseménye legyen a következő az eseménysorban, és a kiinduló állapotának szülő állapota aktív legyen (in_PowerOn). Tüzelés után beállítjuk az akciót

(DisplayMaxSignal), és jelezzük, hogy a LineOK állapot aktív lett. Erre azért van szükség, mert a modell másik részében erre egy őrfeltételben hivatkozni kell.

Az Uppaal a modellek leírására egy saját XML formátumot használ, melynek adott a sémája. A következő példa egy átmenetet ír le, sorra megadjuk a forrás és cél állapotot, a tüzeléskor végrehajtandó akciót, a szinkronizációt, végül egy töréspontot (nail), melyet az átmenet ábrázolásakor használ fel az Uppaal.

```
<transition>
      <source ref="id0"/>
      <target ref="id1"/>
      <label kind="assignment" x="300" y="280">
            event2mobile := Disconnected
      </label>
      <label kind="synchronisation" x="300" y="270">
            mobileDispatcherWrite!
      </label>
      <nail x="300" y="290"/>
</transition>
```

A modellhez teljes állapotlefedést biztosító teszteket generáltam. Az ellenőrizendő tulajdonságokban beépített konstrukcióval lehet hivatkozni az egyes processzek állapotára, így az állapotfedést vizsgáló formulák egyszerűen megadhatók. Sajnos átmenetek esetén nem ilyen szerencsés a helyzet, erre nincsen beépített elem. Így minden egyes átmenet akcióját bővíteni kell, ahol egy globális változóba beállítjuk, hogy épp melyik átmenet tüzelt, és ezt lehet az ellenőrzés során figyelni.

A generáláshoz szélességi bejárást alkalmaztam, és a –t 1 kapcsolóval trace információkat is kiírattam, mégpedig a legrövidebb ellenpéldához tartozót. Ez ugyan letiltja az adatstruktúrák újrahasznosítását az egyes tulajdonságok ellenőrzése között, azonban a modell kellően kicsi volt ahhoz, hogy ez ne okozzon gondot.

A kiadódó ellenpélda leírásában könnyű volt az eseményeket és az akciókat kiszűrni, az event2mobile és nextEvent4mobile változók jelzik, hogy mit küldtek az állapottérképnek, és az mit olvasott ki éppen az eseménykezelőből, az outputEvent pedig a válaszként generált akciókat tartalmazza.

Láthattuk, hogy a korábbi fejezetekben vázolt tesztgenerálási módszer valósidejű rendszerek esetén is működik. A SPIN esetén felmerült feladatok, legrövidebb futás kezelése, formulák generálása, kimenet feldolgozása itt is kezelhetőek. A tesztgenerálás automatizálásához azonban el kell készíteni itt is egy bizonyítottan helyesen működő transzformációt, azonban ez túlmutat ezen diplomamunka keretein.

8 Konklúzió

A diplomamunka elkészítése során kidolgoztam egy keretrendszert modell alapú automatikus tesztgenerálás támogatására. A diplomaterv áttekintette a beágyazott rendszerek modellezésével, ellenőrzésével és tesztelésével kapcsolatos ismereteket. Ezután megvizsgálta a szakirodalomban megjelent, modell alapú teszteléssel kapcsolatos fontosabb publikációkat. Külön kitért a jelenleg elérhető akadémiai és kereskedelmi eszközökre, egyúttal rávilágított a nyitott kérdésekre.

A diplomamunka részletesen bemutatott egy módszert, melynek segítségével automatikusan lehet állapottérképekhez teszteket generálni. A folyamat modell ellenőrző eszközt használ a modell bejárásához és a teszteket reprezentáló eseménysorozatok megkereséséhez. A dolgozat megvizsgálta a módszer elméleti hátterét, majd a tesztelésnél használt fontosabb fedési kritériumok temporális logikai megfelelőjét ismertette, melyek a modell ellenőrző egyik bemenetét képezik. Bemutatott továbbá egy transzformációt, mely UML állapottérképeket a SPIN modell ellenőrző nyelvére alakítja.

A tanulmány az elkészült tesztgenerátor implementálásával kapcsolatos tapasztalatokkal folytatódott. A negyedik fejezet tartalmazta a megvalósult szoftver specifikációját, UML diagramokkal megadott statikus és dinamikus modelljét és a Java nyelvű implementáció fontosabb elemeit. A program parancssoros és grafikus felületet is biztosít a többféle kritérium alapján történő tesztgenerálásra.

Az ötödik fejezet a generált teszteken végzett mérésekről és egy valós alkalmazhatóságot bizonyító esettanulmányról számolt be. A SPIN modell ellenőrző sokféle validációs technikát ismer, és szerteágazóak a paraméterezési lehetőségei is, azonban a mérésekkel sikerült azonosítani azt a néhányat, mely hatékony futást eredményez a tesztgenerálás speciális igényei esetén is. A szinornizációs protokoll kapcsán láthattuk, hogy még igen nagyméretű modell esetén is használható az eszköz, ehhez azonban további, az állapottér tömörebb reprezentációját lehetővé tevő módszert kellett alkalmazni.

A hatodik fejezet bebizonyította, hogy a generált tesztek tovább alakíthatóak úgy, hogy ne csak a modell, hanem az ahhoz készült implementáció ellenőrzésére is felhasználhatóak legyenek. Példákon keresztül bemutatta, hogyan lehet Rational Robothoz és JUnithoz illeszteni a teszteket, viszonylag kevés további implementáció specifikus kód hozzáadásával. A generált tesztek „jóságát” az implementáción elért kódfedési eredmények demonstrálták. 85-95%-os utasítás lefedettséget sikerült elérni, mely nem biztonságkritikus rendszerek estén jónak mondható. A fedettséget tovább lehet növelni a hiányzó részeket direktben tesztelő saját formulákkal vagy pedig, mivel a kimaradt kódsorok nagy része hibakezelő rutin, implicit átmeneteket lefedő teszt esetekkel.

A módszer kiterjeszthető valós idejű rendszerekre is. A hetedik fejezet ismertette ennek lehetőségeit és egy mintapéldán keresztül bemutatott egy lehetséges megvalósítást is. A példák segítségével sikerült azonosítani a fontosabb problémákat,

melyekkel egy, az automatizáláshoz szükséges transzformáció tervezése és implementálása esetén szembe kell nézni.

A program főleg vezérlésorientált, beágyazott rendszerekhez használható, ugyanis az alkalmazott transzformáció jelenleg egyszerű jelzések (signal) és események küldését kezeli, az állapottérképeken lévő változókat illetve komplex adatstruktúrákat egyelőre nem támogatja.

Az általam elért eredményeket röviden tehát így lehetne összefoglalni:
1. Új tesztgeneráló eszköz elkészítése.
2. A SPIN modellellenőrző különböző konfigurációs lehetőségeinek vizsgálata tesztgenerálási szempontból.
3. Nagyméretű állapottérképen végzett tesztgenerálás.
4. Modell alapján generált tesztek illesztése az elterjedt teszt futtató keretrendszerekhez.
5. Alkalmazási korlátok és kiterjesztési lehetőségek vizsgálata.

Tudomásom szerint nem publikáltak még olyan eredményeket, amelyek hasonló méretű UML állapottérképből való automatikusan tesztgenerálásról szóltak volna. Offutték eszköze sokkal szűkebb állapottérkép osztállyal dolgozik [OA99], a [HIM00]-ban használt példa jelentősen kisebb komplexitású, a GOTCHA eszköz nagyméretű modelleket használt ugyan, de azokat a saját nyelvén kellett megadni. A telekommunikációs iparágban is alkalmaztak nagyobb modelleket, viszont azok jellemzően SDL leírásúak, és konformancia tesztelésükre léteznek kidolgozott eljárások.

A diplomamunkában bemutatott módszer és eszköz sokféle továbblépési irányt kínál. A kiinduló transzformáció elemkészletének bővítésével, az állapottérképen definiált változók kezelésével további, adat alapú kritériumok implementálása is lehetővé válik. A generált tesztek utólagos feldolgozásával is részben ki lehet szűrni a felesleges eseményeket, melyek nem generálnak akciókat, így valószínű, hogy kevésbé erőforrás-igényes ellenőrzéssel is kaphatók minimális hosszú tesztek. A tesztkészlet összeállításánál érdemes lehet megvizsgálni, hogy nem minimális hosszúságú tesztek segítségével mennyivel nagyobb kód fedést lehet elérni és ehhez mekkora futási idő növekedés tartozik. Ezen cél elérése érdekében további, lehetőleg valós állapottérképeken szükséges méréseket végezni.

A mérések során használt két példával demonstráltam a módszer és az eszköz alkalmazhatóságát, azonban még –mint arra több helyen is utaltam– sokféle továbblépési lehetőség kínálkozik.

Irodalomjegyzék

[AB02] Paul Ammann, Paul E. Black, and Wei Ding, Model Checkers in Software Testing, NIST-IR 6777, NIST, 2002

[ABM98] Paul E. Ammann, Paul E. Black, and William Majurski, Using Model Checking to Generate Tests from Specifications, Proceedings of ICFEM'98, Brisbane, Australia (December 1998)

[AGEDIS] AGEDIS projekt, http://www.agedis.de/

[AsmL] Foundations of Software Engineering – AsmlL http://research.microsoft.com/fse/asml/default.aspx

[ATG] Automatic Test Generation from Formal Specifications, http://hissa.nist.gov/~black/FTG/autotest.html

[ATGT] ATGT: ASM Tests Generation Tool: http://www.dmi.unict.it/garganti/atgt/

[Beck04] Kent Beck: Extreme Programming Explained, Addison-Wesley, Paperback, 2nd edition, Published November 2004, ISBN 0321278658

[Binder99] R. Binder: Testing Object-Oriented Systems, Addison-Wesley, 1999

[BR04] Bhaduri, Purandar; Ramesh: Model Checking of Statechart Models: Survey and Research Directions, eprint arXiv:cs/0407038, 07/2004

[Clover] Cenqua Clover, http://www.cenqua.com/clover/

[Dav03] Jim Davies: Test Suites from Object Models, ACM SAC Conf., 2003.

[DMY01] A. David, O. Möller, W. Yi: Formal Verification of UML Statechart with Real-time Extensions, in Proceedings of the Nordic Workshop on Programming Theory, 2001

[EFM97] Andre Engels, Loe Feijs, Sjouke Mauw: Test Generation for Intelligent Networks Using Model Checking, Proceedings of the TACAS '97

[FHNS02] Galit Friedman, Alan Hartman, Ken Nagin, and Tomer Shiran: Projected state machine coverage for software testing, ISSTA 2002.

[GNTV03] W. Grieskamp, L. Nachmanson, N. Tillmann and M. Veanes: Test Case Generation from AsmL Specifications, Extended Abstract of work in progress for ASM2003

[Gries03] Grieskamp et al: Model-Based Testing with AsmL .NET, 1st European Conference on Model-Driven Software Engineering, Dec. 11-12, 2003

[GRR03] A.Gargantini, E.Riccobene, S.Rinzivillo. Using Spin to Generate Tests from ASM Specifications. ASM03, LNCS 2589, 2003.

[Hart03] Alan Hartman: Model-based test generation tools, www.agedis.de/documents/ModelBasedTestGenerationTools_cs.pdf

[HIM00] Jean Hartmann, Claudio Imoberdorf, Michael Meisinger: UML-Based integration testing, Proceedings of the 2000 ACM SIGSOFT, 2000

[HLSC01] Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, Sung Deok Cha: Automatic Test Generation from Statecharts Using Model Checking, Technical Report MS-CIS-01-07, Feb 2001.

[HLSCU03] Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, Sung Deok Cha, Hasan Ural: Data flow testing as model checking, International Conference on Software Engineering, ISBN: 0270-5257, 2003.

[Holz97] Gerard J. Holzmann: The Model Checker Spin, IEEE TSE (23)5, pp. 279-295, 1997.

[Holz03] Gerard J. Holzmann: The Spin Model Checker: Primer and Reference Manual, Addison-Wesley, ISBN 0-321-22862-6, 2003

[JUnit] http://www.junit.org

[Leg02] B. Legeard et al : BZ-Testing-Tools: A Tool-Set for Test Generation from Z and B using Constraint Logic Programming, In proc. of FATES'02, Formal Approaches to Testing of Software, 2002

[LMM99] D. Latella, I. Majzik, M. Massink: Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker, Formal Aspects of Computing, Volume 11 Issue 6 (Springer Verlag) pp 637-664, 1999.

[McM93] K. McMillan: Symbolic Model Checking. Kluwer Academic, 1993.

[MTG] Model-based Test Generation, http://www.cis.upenn.edu/~rtg/testgen/

[OA99] Jeff Offutt and Aynur Abdurazik: Generating Tests from UML Specifications, UML99, Fort Collins, CO, October 1999.

[Pat03] Pataricza András et al: Formális módszerek az informatikában, Typotex Elektronikus Kiadó Kft., 2003

[PM03] Gergely Pintér, István Majzik: Program code generation based on UML Statechart models, Periodica Polytechnica Ser. El. Eng. Vol. 47, No. 3-4, pp. 187-204 (2003)

[RAO92] D. Richardson, S. Aha, and T. O'Malley, "Specification-based test oracles for reactive systems," in Proc. Fourteenth Int. Conf. Software Eng., Melbourne, Australia, May 1992.

[Robot] IBM Rational Robot User's Guide, G126-5388-00, http://www-1.ibm.com/support/docview.wss?uid=pub1g126538800

[Spin] Spin Online References, http://www.spinroot.com

[TestManager] IBM Rational TestManager User's Guide, G126-5385-00 http://www-1.ibm.com/support/docview.wss?uid=pub1g126538500

[UML03] Unified Modeling Language, specification v1.5 http://www.omg.org/cgi-bin/doc?formal/03-03-01

[UPPAAL] Paul Pettersson and Kim G. Larsen: Uppaal2k, in Bulletin of the European Association for Theoretical Computer Science, volume 70, pages 40-44, 2000.

Köszönetnyilvánítás

Ezúton szeretném megköszönni konzulensemnek, Dr. Majzik Istvánnak a felkészülés és a diplomaterv megírása során nyújtott segítségét, aki értékes észrevételekkel és rengeteg hasznos tanáccsal járult hozzá tanulmányom elkészítéséhez.

Köszönetemet fejezem ki Pintér Gergőnek, aki rendelkezésemre bocsátotta a kódgenerátor programját, és segített, hogy a teszt illesztéssel és kódfedéssel kapcsolatos mérésekben fel tudjam használni.

Szívből köszönöm Telek Andrea stilisztikai és formai kérdésekkel kapcsolatos tanácsait és segítségét.

Végül, de nem utolsó sorban, köszönöm szüleim gondoskodását és folyamatos támogatását, mellyel lehetővé tették, hogy eljussak idáig.

Függelék

A. Generált Promela kód

Egy egyszerű állapottérkép transzformált Promela kódja, a lényegesebb részeket kiemelve:

```
/* eseményekhez számokat rendelünk */
#define goA 1
#define goD 2

/* az objektumok üzenetsora, ha valaki küldeni akar nekik egy
eseményt,
   akkor ebbe ír
*/
chan object1_queue = [2] of {int};

/* bitek, melyek jelzik, hogy az egyes átmenetek tüzelhetőek-e */
bit Cand_object1_t43, Cand_object1_t44;

/* bitek, melyek jelzik, hogy az egyes objektumok az adott
állapotban
   vannak-e
*/
bit object1_A, object1_B;

/* ennek a számlálónak a növelésével jelzi az objektum,
   hogy a dispatcher válasszon ki a számara egy eseményt a sorból
*/
int disp_object1_queue_counter=0;

/* az objektum és a dispatchere között lévő csatorna */
chan disp_object1 = [0] of {int};

/* dispatcher, feladata, hogy kiválasszon egy eseményt a sorból */
proctype dispatcher_object1_queue()
{
  int Ev;
  do
  ::  (disp_object1_queue_counter == 1);
      disp_object1_queue_counter = 0;
      object1_queue?Ev;
      disp_object1!Ev;
  od
}

/* az object1 objektum állapottérképének egy lépését végző
processz */
proctype STEP_object1()
{
  int Ev=0;
  bool completed=false;

  do
  ::if
    /* UML run-to-completion elv, addig nem kér majd új eseményt,
       amíg tud lepni (van tüzelhető trigger esemény nélküli
```

```
          átmenet)
       */
       :: (completed) -> disp_object1_queue_counter++;
          disp_object1?Ev; completed=false;
       :: else -> Ev=0;
       fi;
       atomic
       {
         /* megvizsgáljuk, hogy melyik átmenet tüzelhető =
            engedélyezett (a trigger eseménye érkezett, teljesül az
            őrfeltétele)+ nincs nála nagyobb prioritású engedélyezett
          */
         Cand_object1_t43 = (object1_A);
         Cand_object1_t44 = (object1_B & (Ev==goA));

         /* a tüzelhető halmazokból véletlenszerűen választunk egyet
            Megj.: ebben a példában nincsen konkurens régió, így a
            tüzelhető halmazok egy eleműek. Ha lenne, akkor minden
            régiónak megfelelne egy object_AUTOXX processz, amiket
            ilyenkor léptetne a megfelelő felső szintű átmenet
            tüzelése
          */
         if
         :: Cand_object1_t43 -> object1_A=0; object1_B=1;
                                object2_queue!goD;
         :: Cand_object1_t44 -> object1_B=0;object1_A=1;
         :: else ->             completed=true;
         fi;
       }
     od
}

init
{
  atomic
  {
    /* kezdőállapotokat beállítjuk */
    object1_A=1; object1_B=0;

    run STEP_object1();
    run dispatcher_object1_queue();
  }
}
```

A ! F (object1_A) LTL formulából generált never claim:

```
#define in_object1_A (object1_A == 1)
never {    /*  (<>object1_A) */
T0_init:
     if
     :: ((object1_A)) -> goto accept_all
     :: (1) -> goto T0_init
     fi;
accept_all:
     skip
}
```

B. Teszteset

Egy generált, XML formátumú teszteset (rövidített változat):

```
<?xml version="1.0" encoding="UTF-8"?>
<testcase>
     <covering>
          <transition event="GET" guard="" from="GetSent"
           to="GetSent" automaton="HeadA">
              <action name="aGETACK_INV"/>
          </transition>
     </covering>
     <events>
          <event action="send" from="init" to="HeadA" name="GET"/>
          <event action="send" from="init" to="HeadB" name="GET"/>
          <event action="receive" from="" to="HeadB" name="GET"/>
          <event action="receive" from="" to="Timer" name="GET"/>
          <event action="receive" from="" to="Timer" name="clock"/>
          <event action="receive" from="" to="HeadA" name="GET"/>
     </events>
</testcase>
```

C. JUnit teszt generálása

A nested switch módszerrel megvalósított kód esetén a JUnit tesztek generálásához szükséges két legfontosabb sablon.

Esemény küldő sablon:

```
## parameter %EVENTNAME% - the event to dispatch
mobile.handleEvent( EventType.stringToInt( "%EVENTNAME%" ) );
System.out.println("Action: " +
EventType.intToString(actionReceived));
```

Akció ellenőrző sablon:

```
## parameter %ACTIONNAME% - the action sent as a response
assertEquals( EventType.%ACTIONNAME%, actionReceived );
```

A sablonok segítségével generált Java kód (a // # formátumú megjegyzések közötti részek azok, amiket a kézzel kell megírni az adott implementációnak megfelelően):

```
/* Auto-Generated by generateJunitTest */
package testGenerator.examples.mobile;
import junit.framework.*;
import java.io.*;

public class MobileNestedSwitchStateTest extends TestCase
implements ActionEventListener
{
    // # ------- Private fields -------
    private MobileNestedSwitch mobile = null;
    private int actionReceived = -1;
    // # ------- Private fields -------

    // # ------- SUT specific methods -------
    public void actionEventOccured( ActionEvent event )
    {
        actionReceived = event.getAction();
    }
    // # ------- SUT specific methods -------

    public MobileNestedSwitchStateTest(String testName)
    {
         super(testName);

         // # ------- Custom constructor code -------
         mobile = new MobileNestedSwitch();
         mobile.addActionEventListener( this );
         // # ------- Custom constructor code -------
    }

    protected void setUp() throws Exception
    {
         // # ------- Setup code -------
         mobile.init();
         // # ------- Setup code -------
    }
```

```
public static Test suite()
{
    TestSuite suite = new TestSuite(
         MobileNestedSwitchStateTest.class );
    return suite;
}


// ------- Test methods -------

/** Test for mobile.CallWait state */
public void testInMobileCallWait()
{
     System.out.println("InMobileCallWait test");
     mobile.handleEvent(EventType.stringToInt("evKeyNoHold"));
     System.out.println( "Action received: " +
          EventType.intToString( actionReceived ) );
     assertEquals( EventType.OpenDisplay, actionReceived );

     mobile.handleEvent(EventType.stringToInt("evCalling" ));
     System.out.println( "Action received: " +
          EventType.intToString( actionReceived ) );
     assertEquals( EventType.RingOn, actionReceived );
}
```

D. Ant Build.xml

Egy tipikus Ant leíró fájl a Clover fedettséget vizsgáló műveleteivel.

```
<project name="clover.generated" default="main" basedir=".">
    <!-- a build folyamán előállított összes fájl helye -->
    <property name="build.dir" value="${basedir}\build" />
    <property name="reports.dir" value="${build.dir}\reports"/>
    <property name="src.dir" value="${basedir}\src" />
    <!-- hivatkozás a clover által definiált egyéni taskokra -->
    <taskdef resource="clovertasks" />
    <!-- clover inicializáló task -->
    <target name="with.clover">
        <clover-setup initString="mycoverage.db" />
    </target>

    <!-- html jelentés készítése a begyűjtött adatokból -->
    <target name="clover.html" depends="with.clover">
        <clover-report>
            <current outfile="${reports.dir}\clover_html">
                  <format type="html" />
            </current>
        </clover-report>
    </target>
    <!-- a teljes build folyamat -->
    <target name="main"
      depends="clean,init,with.clover,compile,test,clover.html"/>

    <!--  inicializálás, könyvtárak létrehozása -->
    <target name="init">
        <mkdir dir="${build.dir}" />
        <mkdir dir="${build.classes.dir}" />
        <mkdir dir="${reports.dir}" />
    </target>

    <!-- forrásfájlok lefordítása -->
    <target name="compile">
     <javac srcdir="${src.dir}" destdir="${build.dir}/classes" />
    </target>

    <!-- tesztelés JUnit segítségével  -->
    <target name="test" description="Unit test the application">
        <junit fork="yes" dir="${basedir}">
            <classpath>
               <pathelement path="${ant.home}/lib/clover.jar" />
            </classpath>
            <classpath location="${build.classes.dir}"/>

            <formatter type="xml" />
            <batchtest todir="${reports.dir}" >
                  <fileset dir="${src.dir}">
                        <include name="**/*Test.java" />
                  </fileset>
            </batchtest>
        </junit>
    </target>
</project>
```