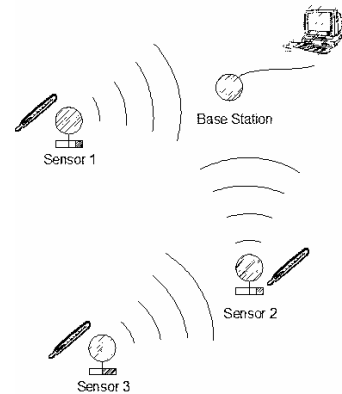


SZENZORHÁLÓZATOK

A TinyOS operációs rendszer és
a Berkeley MOTE platform
bemutatása

Völgyesi Péter
volgyesi@mit.bme.hu



A beágyazott rendszerek egyik jellegzetes és sok új kihívást jelentő csoportja a *szenzorhálózatok* világa. Szenzorhálózatnak nevezzük nagy számú független (*autonóm*) intelligens érzékelőkből alkotott kooperatív hálózatot, ahol az egyes érzékelők valamilyen közös feladat végrehajtását elosztott módon valósítják meg. Az elosztott működés oka a legtöbb esetben az, hogy a fizikai rendszer – melyet mérni vagy befolyásolni szeretnénk – térbeli kiterjedése nem teszi lehetővé, hogy egyetlen központi eszközzel valósítsuk meg a feladatot. A hálózati kapcsolatok (melyek gyakran kis sáv szélességű rádiós csatornákat jelentenek) korlátai teszik szükségessé az adatok minél magasabb szintű feldolgozását az érzékelők szintjén. Tipikus elrendezésben az érzékelők különböző környezeti paramétereket mérnek (fényerő, hőmérséklet, légnyomás, stb.), és a mért értékeket vagy az azokból származtatott mennyiségeket egy központi csomópont – a *bázisállomás* – felé továbbítják.

A fentiek alapján sejthető, hogy a hálózatot alkotó autonóm elemek a legtöbb esetben rendelkeznek saját CPU-val, memóriával (mikrokontroller), saját energiaforrással, különböző érzékelőkkel és A/D átalakítóval, kommunikációs interfészekkel – pl. rádió, soros vonali interfész, valamilyen „field bus” – és néhány esetben beavatkozó eszközökkel.

Szenzorhálózatok jellegzetességei

Ahogy a bevezetőben utaltunk rá, a szenzorhálózatok – a beágyazott rendszereknél eddig tapasztalt problémák mellett – új kihívásokat adnak. A legfontosabb megoldandó feladatok ill. nehézségek:

- **Erőforrás korlátok:** az autonóm működésből következő egyik legfontosabb korlát az energiaforrás véges – és szűkös – kapacitása, továbbá a fizikai méret, a rendelkezésre álló memória (ez sok energiát fogyaszt), a működési, adatfeldolgozási sebesség
- **Konkurencia:** egyrészt a szenzoron belüli párhuzamos feladatok végrehajtása (paraméterek mérése, adatok feldolgozása, kommunikáció a szomszédokkal) jelent problémát, másrészt a szenzorhálózat – mint rendszer – jellegénél fogva magas szintű konkurens (elosztott) viselkedést mutat, mely a hibakeresést és szimulációt megnehezíti
- **Többféle architektúrális megoldás:** mind a paraméterek feldolgozása mind a kommunikációs protokollok terén számos megvalósítási alternatíva létezik; az eszközökön futó operációs rendszernek támogatnia kell a legkülönbözőbb architektúrális megoldásokat, miközben a memória ill. az energiaforrások mérete szűkös – így például egy adott konfiguráció nem tartalmazhatja az összes alternatívát egyetlen monolitikus csomagként (ahogy azt sok általános célú operációs rendszer teszi)

- **Osztott és megbízhatatlan kommunikációs csatornák:** sok esetben vezeték nélküli (rádió, infravörös fény, esetleg hanghullámok) kommunikációs csatornákat használunk. Mivel ezek átviteli közegét egyszerre használja az adott területen található összes érzékelő, megoldandó feladat a csatornához történő hozzáférés szabályozása. A vezeték nélküli átvitel további problémája a vezetékes megoldásokkal szemben a jóval gyakoribb tévesztés a nagyobb hibaarány. Sok esetben az érzékelők helye nem előre rögzített ill. a használat során egyes érzékelők elromolhatnak, így a kommunikációs (logikai) topológia – pl. útvonalválasztás – ad-hoc módon jön létre ill. változik működés közben.

A szenzorhálózatok működésének ill. feladatainak jobb megértése végett lássunk néhány példát.

Aktív zajcsökkentés

Egy, a BOEING cég által vezetett projekt a hordozórakéták fellövésekor jelentkező rázkódás csökkentését tűzte ki célként. A rakéta konténerében (fairing) ill. annak felületén jelentkező rázkódás sokszor jelentős károkat okoz a rakományban (pl. műhold), ennek a rázkódásnak akár részleges csökkentése is jelentősen növeli a sikeres fellövés valószínűségét.

A konténer felületén – előre meghatározott topológia szerint – piezo-elektromos érzékelőket ill. beavatkozó elemeket helyeztek el. Az érzékelők a környezetükben mért rezgést – pl. annak pillanatnyi amplitúdóját – figyelembe véve olyan beavatkozó jelet generálnak, hogy az kioltsa az eredeti rezgést. Jó eredmények akkor születnek, ha az érzékelők egymáshoz is eljuttatják a mért értékeket, és így az egyes beavatkozó jelek meghatározásakor nem csak a lokális értékek állnak rendelkezésre.



1. ábra A rakéta konténere, az ún. fairing

A projektben használt érzékelők között nagy sávzélességű vezetékes hálózatot építettek ki, melyen CORBA alapú kommunikációt használtak. Bár a zajos és szűkös csatornákra tett korábbi megjegyzésünk itt nem igaz, a feladatot nehezítette az erősen valós idejű követelmények betartása – hiszen az elkésett beavatkozás nemhogy felesleges lenne már, hanem kifejezetten káros is lehet. A kísérleti rendszer „frekvenciája” 2kHz volt: a mért értékek továbbítását, feldolgozását ill. a vezérlő jelek előállítását ezzel a frekvenciával kellett megoldani az egyes érzékelőkön ill. a teljes rendszeren.

Környezeti paraméterek megfigyelése

Az amerikai Maine állam partjainál fekvő Great Duck Island sziget mikroklímájának folyamatos megfigyelését tűzte ki célul a kaliforniai Berkeley Egyetem csapata (<http://www.greatduckisland.net>). Mivel ökológiailag törekeny területről van szó, olyan monitor rendszert kellett kialakítani, mely minél kevesebb emberi jelenlétet igényel, és a környezetre gyakorolt hatása is minimális.

Az érzékelőket ill. az azokon futó operációs rendszert – melyekkel később részletesen is meg fogunk ismerkedni – a Berkeley Egyetem oktatói és hallgatói készítették. Ezekre már sokkal inkább igazak azok az erőforráskorlátokra tett megjegyzések, melyekről korábban szó volt. A szigeten szétszórta kis méretű szenzorok hosszú időn keresztül kell, hogy mérjenek olyan paramétereket, mint hőmérséklet, fényerő, páratartalom ill. a szenzor előtt elhelyezett fészek „aktivitása” (ez utóbbit a fészek és az átlaghőmérséklet különbsége alapján becslik). A passzív érzékelők a mért adatokat egy hierarchikus topológia szerint a szigeten levő bázisállomáshoz továbbítják, ahonnan a mért értékek műholdas kapcsolaton keresztül az interneten is elérhetővé válnak.



2. ábra A Berkeley MOTE platform

A szenzorok áramforrásaként két ceruzaelem szolgál, melyeknek hónapokon keresztül biztosítaniuk kell az érzékelők ill. a rádiós áramkörök energiaellátását. Egy ilyen – védőcsomagolásban levő – autonóm érzékelőt láthatunk a 2. ábrán.

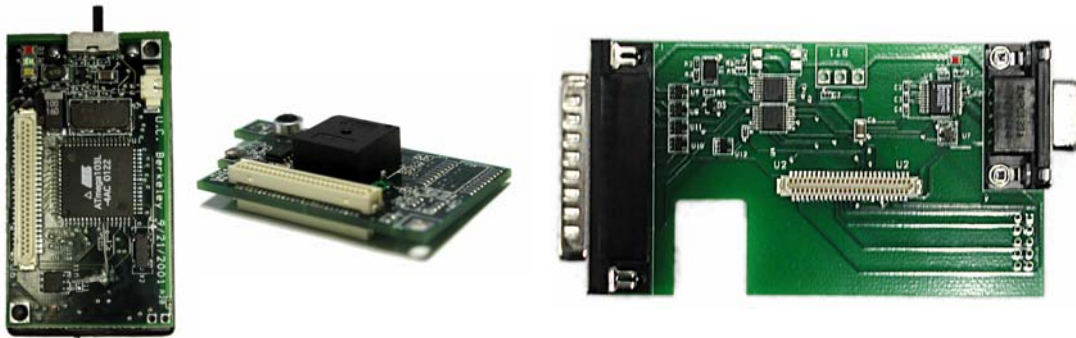
Lőfegyver helyének meghatározása

A DARPA által támogatott NEST projektben a rádiós érzékelőket – melyek a Berkeley MOTE platform változatai – arra használják fel, hogy egy elsütött lőfegyver ill. támadó helyét minél pontosabban meghatározzák (<http://www.isis.vanderbilt.edu/projects/nest>). Ehhez az érzékelőket a védett területen egyenletesen – de nem rögzített topológia szerint – szétszórják. Egy-egy érzékelő több mikrofonnal rendelkezik, melyek jeleit párhuzamosan képes feldolgozni. Így, mivel az egyes mikrofonokhoz a robbanás okozta hanghullámok időben kissé eltérő időbélyeggel érkeznek (ez az időkülönbség a mikrofonok között levő néhány centiméteres távolság miatt jelentkezik), az érzékelők egyenként képesek meghatározni a támadó irányát. Ezeket az irányokat továbbítva a központ felé a lövés helye pontosan meghatározható – ez az irányok által kijelölt egyenesek metszéspontjában lesz.

A projektben az ad-hoc topológia ill. az „egyszerre jelentkező” rádiós kommunikáció kellemtelenségei jelentenek igazi kihívást a szenzorok elvégzendő jelfeldolgozás mellett.

A Berkeley MOTE platform

A példaként hozott projektek közül kettőben az ún. MOTE platformot használták (<http://webs.cs.berkeley.edu/tos>). Ezek az eszközök viszonylag kis méretük és nyílt forráskódú fejlesztésük miatt rendkívül alkalmasak kísérleti rendszerek elkészítéséhez ill. oktatási célú felhasználáshoz. Röviden áttekintjük a platform hardver jellemzőit – elsősorban a korlátok érzékeltetése végett –, majd bővebben tárgyaljuk az eszközöket működtető operációs rendszer architektúráját.



3. ábra A MOTE alaplap, a szenzor kártya és a programozó

A platform moduláris felépítésű: az alaplap tartalmazza az áramforrást (2db AA elem), a központi feldolgozó egységet (ATmega 128 mikrokontroller), néhány LED kijelzőt, a rádiós áramkört és külső flash memóriát. A felhasznált RISC architektúrájú mikrokontroller 4Mhz-es órajelen jár, 128kB belső flash memóriával, 4kB SRAM adatmemóriával továbbá valós idejű órával, 8 A/D csatornával, soros (UART) és SPI interfészekkel rendelkezik. A mikrokontroller ill. a MOTE platform támogatja a rendszerben történő programozást (*In System Programming*), így új program feltöltéséhez a mikrokontrollert nem kell kivennünk az eszközből, ezt a 3. ábrán is jól kivehető csatlakozón keresztül a programozó kártyával végezhetjük el. Ugyanezen a csatlakozón keresztül – a programozó kártya segítségével – csatlakoztathatjuk az érzékelőt a PC soros portjára.

A rádiós kommunikáció 900Mhz-en zajlik a vivő jel ki ill. bekapcsolásával (*on-off keying*), az elérhető „nyers” átviteli sebesség 50kBit/s, a hatótávolság pár méter. Fontos megjegyezni, hogy a rádiós áramkör bit szintű felületet nyújt, a csomagok keretezését, a bitek kódolását valamint a csatorna hozzáférés szabályozását mind szoftverből kell megoldani.

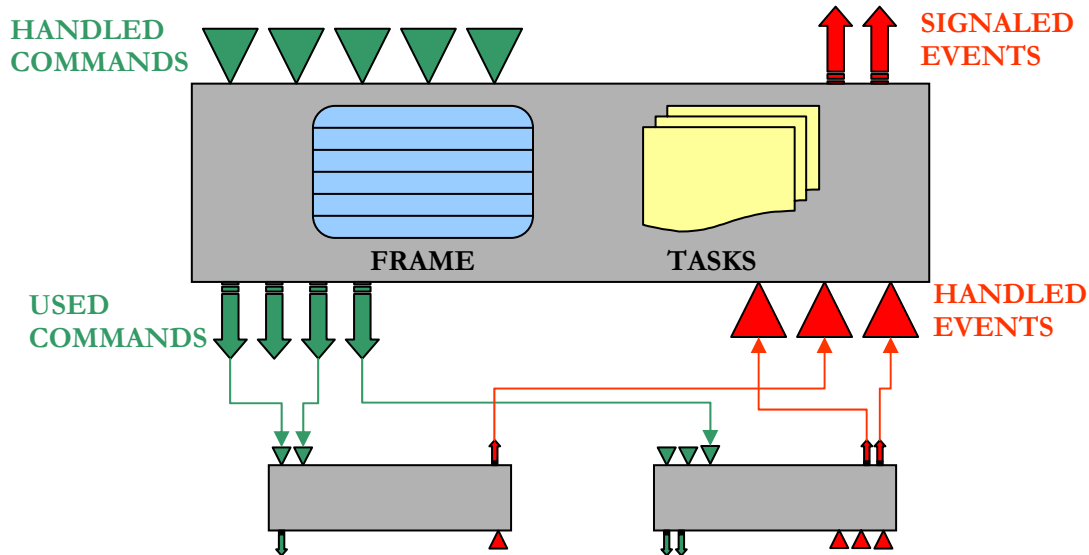
Az említett modularitás jegyében az érzékelők külön lapra kerültek, így különböző konfigurációk alakíthatók ki különböző célokra. Az ábrán látható általánosnak tekinthető konfiguráció tartalmaz egy mikrofont, egy „buzzer”-t, hőmérőt, fénymérőt, gyorsulás és mágneses teret érzékelő eszközöket. Az érzékelők mindegyike szoftverből kikapcsolható, amit programjainknak ki is kell használniuk.

A TinyOS operációs rendszer

A MOTE platformot működtető *TinyOS* operációs rendszer egyszerre célozza meg a moduláris tervezés és a hatékonyság (elsősorban kis méret és optimális kód) előnyeit. A beágyazott alkalmazást önálló – szigorú interfészekkel elválasztott – szoftverkomponensek hálózatának tekinti. A komponensek „hálózatát” fordítási időben alkotjuk meg, később ezek a kapcsolatok nem módosulhatnak, a rendszer a futási időben történő dinamikus átkonfigurálást nem támogatja. Az operációs rendszer előre elkészített komponensek halmaza, melyekből tetszőlegesen válogathatunk, köthetünk saját komponenseinkhez. A hálózat kialakítása (szerkesztés – linker) után azonban egy olyan monolitikus program jön létre, melyet egyben lehet csak a hardverre

letölteni. Fontos megjegyezni, hogy az operációs rendszer mindig része az alkalmazásunknak, a hardveren nincsen permanensen futó mag, ahogy azt például az általános célú számítógépeken megszoktuk.

A TinyOS környezet támogatja a többtaszkos végrehajtást, de a taszkok nem szakíthatják meg egymást (nem preemtív az ütemezés). Dinamikus memóriefoglalás nincsen, de tekintve a rendelkezésre álló néhány kB kapacitást ennek nincs is gyakorlati jelentősége.



4. ábra Szoftverkomponens a TinyOS rendszerben

Egy jellemző szoftverkomponenst és annak kapcsolatait mutatja a 4. ábra. Egy szoftverkomponens nem más, mint egy C nyelven írt forrásfájl, mely statikus adatterületek ill. függvények halmaza. A komponens statikus adatterületeit fogja össze a TinyOS egy speciális struktúrába, az ún. *frame*-be. A frame-ben lévő adatokat csak a komponens használhatja, azok kívülről nem elérhetők közvetlen módon.

A C fájlban definiált függvények egy része a „felsőbb” komponensek felől érkező kérések kiszolgálására íródik, ezek a parancskezelők (*command handlers*), másik részük – az eseménykezelők (*event handlers*) – az „alsóbb” komponensek által jelzett eseményekre reagálnak. A két függvénytípus megvalósítása között érdemi eltérés nincs, a különbségtétel pusztán formai. A komponens is használ parancsokat (*commands*) ill. jelezhet eseményeket (*events*), a hagyományos C nyelvtől eltérően azonban a TinyOS rendszerben ezeket a függvényhívásokat is fel kell tüntetnünk a komponens interfészén. A függvényhívásoknál azokat a neveket használjuk, melyet a komponens interfészének definiálásakor választottunk. Az, hogy ezek a hívások mely másik komponenshez fognak beérkezni, később dől majd el a teljes alkalmazás „összehuzalozása” során. Ez az alapja a TinyOS moduláris szerkezetének.

A TinyOS egyik alapszabálya, hogy a parancsok és események kezelése során nem tarthatjuk fogva hosszú ideig a processzort, akár blokkoló jellegű műveletekről, akár időigényes számítási feladatról legyen szó. Ilyen esetekben a kezelő függvénynek egy taszkot kell ütemeznie, melynek során egy – a komponensben megvalósított – függvényre mutató pointert ad át az ütemezőnek. Az ütemező, ha a parancsok és események kezelése megtörtént, előveszi az eddig regisztrált függvénypointereket, és egymás után hívja meg azokat. Egy ilyen taszk tehát nem szakíthatja meg a másik taszk végrehajtását, azonban egy külső esemény okozhatja futó taszk felfüggesztését. Az ütemező a taszkokra mutató pointereket egy FIFO sorban tárolja, nézzük a leegyszerűsített megvalósítást:

```
typedef void (*sched_entry) ();
sched_entry queue[MAX_THREADS];
```

```

clear_queue();
main_init();
main_start();

while(1) {
    sched_entry tp = NULL;
    while (tp = get_next(queue)) { tp(); }
    sleep();
}

```

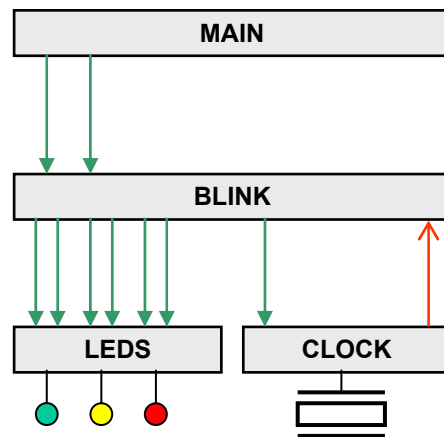
A *queue* nevű tömbben tároljuk a végrehajtandó taszkokra (függvényekre) mutató pointereket. A beágyazott alkalmazás indulásakor töröljük a várakozási sort, majd meghívjuk a *main* nevű komponens két függvényét egymás után. Ez a komponens mindig része az alkalmazásunknak, és további komponensekben található parancsokat hívhat. Ezen parancsok végrehajtása során egyes komponensek regisztrálhatnak később végrehajtandó taszkokat, melyek a queue-ba kerülnek. A parancsokból való visszatérés után az ütemező végtelen ciklusba kezd. Minden egyes iterációs lépésben végigjárja a várakozási sort, és a benne található taszkokat egymás után végrehajtja, majd alvó állapotba teszi a mikrokontrollert. Mivel üres várakozási sor esetén valóban nincs több végrehajtandó feladat, ez így a legtakarékosabb. Egy külső esemény (interrupt) ébresztheti fel majd a kontrollert, mely hatására egy alacsony szintű esemény keletkezik, mely fentebbi eseményeket hívhat. Az eseménykezelők végrehajtása során ismét taszkok kerülhetnek a sorba, melyet az ütemező a következő iterációs lépésben fog végrehajtani.

Beágyazott alkalmazások fejlesztése

A TinyOS szoftverkörnyezet több komponensből áll. A legfontosabb alkotóelemek:

- **avrgcc fordító program:** mivel a fejlesztés PC-n történik, szükség van egy olyan C fordítóra, mely a mikorkontroller gépi kódjának megfelelő bináris állományokat állítja elő. Ez az ún *cross-compiler* a GNU C fordító erre a platformra készített változata
- **cygwin környezet:** erre abban az esetben van szükség, ha Windows operációs rendszerben használjuk a fejlesztő eszközöket (ez a csomag olyan fontos – a Unix világban elterjedt – eszközöket tartalmazza, mint a *make* vagy a *awk*)
- **TinyOS forráskód:** az operációs rendszer előre elkészített és újrafelhasználható komponensei (pl. hálózati kommunikáció, alacsony szintű eszközmeghajtók, a korábban ismertetett ütemező)
- **egyéb eszközök:** ahogy látni fogjuk, a forrásfájlokban egy kicsit módosított C nyelven programozunk. A nyelv kiterjesztése a későbbi „huzalozást” teszi lehetővé. Az előfeldolgozást különböző segédprogramok végzik a forrásállományokon.

A komponensek belső működését mindig egy C fájl segítségével valósítjuk meg. A komponens interfésze – mely tartalmazza a komponensben megvalósított és a komponensből meghívott parancsokat és eseményeket – egy *.comp* kiterjesztésű önálló formátummal rendelkező fájlban kerül leírásra (ebből automatikusan állítja elő az előfeldolgozó a C nyelvben megszokott header állományokat). A komponensek összehuzalozását egy *.desc* kiterjesztésű fájlban adjuk meg. A könnyebb érthetőség végett nézzünk meg egy egyszerű TinyOS alkalmazást, mely adott frekvenciával villogtat egy LED-et az eszközön.



5. ábra A Blink alkalmazás

Az alkalmazás négy komponensből épül fel, melyek közül három a TinyOS operációs rendszer része, a negyedik az alkalmazásfejlesztő által készített modul. A *main* komponenssel már találkoztunk az ütemező ismertetésénél, legfontosabb feladata az alkalmazás inicializálása. A *leds* és *clock* komponensek eszközevélők: az első a mikrokontroller időzítőjét vezérli, utóbbi az eszközön található LED-ek maghajtásáért felelős. Inicializálás során a *blink* komponens felprogramozza az időzítőt periodikus riasztásra, majd a riasztások beérkezésekor a LED-eket ki-ill. bekapcsolja.

Az alkalmazás huzalozását leíró állomány sorai a fenti ábra kommunikációs kapcsolatait (nyilak) határozza meg:

```

include modules {
    MAIN;
    BLINK;
    CLOCK;
    LEDES;
};

BLINK:BLINK_INIT MAIN:MAIN_SUB_INIT
BLINK:BLINK_START MAIN:MAIN_SUB_START

BLINK:BLINK_LEDy_on LEDES:YELLOW_LED_ON
BLINK:BLINK_LEDy_off LEDES:YELLOW_LED_OFF
BLINK:BLINK_LEDr_on LEDES:RED_LED_ON
BLINK:BLINK_LEDr_off LEDES:RED_LED_OFF
BLINK:BLINK_LEDg_on LEDES:GREEN_LED_ON
BLINK:BLINK_LEDg_off LEDES:GREEN_LED_OFF
BLINK:BLINK_SUB_INIT CLOCK:CLOCK_INIT
BLINK:BLINK_CLOCK_EVENT CLOCK:CLOCK_FIRE_EVENT
  
```

Az alkalmazás-komponens (*blink*) interfésze a *.comp* kiterjesztésű fájlban ezek után így néz ki:

```

TOS_MODULE BLINK;

ACCEPTS {
    char BLINK_INIT(void);
    char BLINK_START(void);
};

HANDLES {
    void BLINK_CLOCK_EVENT(void);
  
```

```

};
USES {
    char BLINK_SUB_INIT(char interval, char scale);
    char BLINK_LEDr_on();
    char BLINK_LEDr_off();
    char BLINK_LEDy_on();
    char BLINK_LEDy_off();
    char BLINK_LEDg_on();
    char BLINK_LEDg_off();
};
SIGNALS {
};

```

Az *ACCEPTS* blokk az eseménykezelő függvények prototípusait sorolja fel, a *HANDLES* alatt az eseménykezelőket találjuk, míg a *USES* és *SIGNALS* csoport a meghívásra kerülő parancsok ill. jelzett események gyűjtőhelye.

A blink modul forráskódja ezek után már könnyen érthető. A komponens egyetlen statikus változóval dolgozik, amit a frame területen tárol és a LED állapotát írja le. Az időzítő felől érkező esemény hatására váltogatja a LED állapotát.

```

TOS_FRAME_BEGIN(BLINK_frame) {
    char state;
}
TOS_FRAME_END(BLINK_frame);

char TOS_COMMAND(BLINK_INIT) () {
    TOS_CALL_COMMAND(BLINK_LEDr_off) ();
    TOS_CALL_COMMAND(BLINK_LEDy_off) ();
    TOS_CALL_COMMAND(BLINK_LEDg_off) ();
    VAR(state) = 0
    TOS_CALL_COMMAND(BLINK_SUB_INIT)
        (tick1ps);
    return 1;
}

char TOS_COMMAND(BLINK_START) () {
    return 1;
}

void TOS_EVENT(BLINK_CLOCK_EVENT) () {
    VAR(state) = (VAR(state) + 1) % 2;
    if (VAR(state))
        TOS_CALL_COMMAND(BLINK_LEDr_on)
    else
        TOS_CALL_COMMAND(BLINK_LEDr_off)
}

```

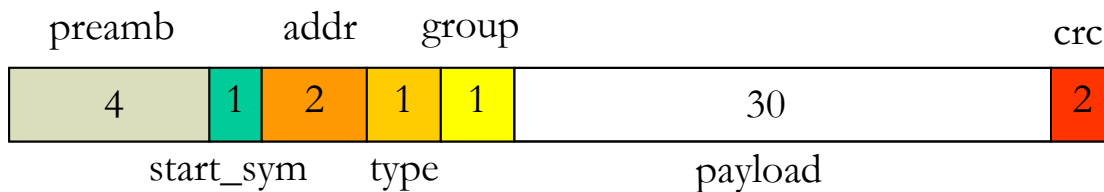
A TinyOS lehetőséget ad komponensek hierarchiába szervezésére is. Így bizonyos komponensek úgy valósíthatnak meg feladatokat, hogy alsóbb komponenseket tartalmaznak. Az alsóbb komponensek rejtve maradnak a fejlesztés során. Lehetőség van virtuális komponensek létrehozására is, melyek a külvilág felé nyújtott szolgáltatásaikat pusztán a tartalmazott alkomponensek összehuzalozásával implementálják.

Hálózati kommunikáció

A TinyOS rendszer egyik legfontosabb – és legbonyolultabb – komponens hálózata a rádiós kommunikációt megvalósító könyvtár. A rádiós adatátvitel szükségessé teszi az elküldendő bitsorozat „kiegyensúlyozottságát” (ne legyen benne egyenáramú komponens, vagyis az 1 és 0 értékű bitek már rövid tartományban is egyenlő számban szerepeljenek). Erre azért van szükség, mert a vevő a vett jel amplitúdóját egy mozgó átlaghoz viszonyítja, és ez alapján dönti el, hogy 1 vagy 0 értéket kapott-e (*on-off keying*). Számos kódolási eljárás használatos, mely kielégíti a fenti feltételt, az egyik legelterjedtebb a Manchester-kód.

A másik leküzdendő probléma, hogy az ütközéseket nem lehet észlelni, és megelőzésük sem teljesen megoldható az átviteli közeg sajátosságai miatt. A TinyOS rendszerben CSMA alapú hozzáférés-vezérlést implementáltak: az adni kívánó csomópont az adás előtt véletlen hosszú ideig vár, majd behallgat a csatornába. Ha adást észlel, ismét véletlen hosszú ideig várakozik, ellenkező esetben elkezd a saját adását. Mivel bizonyos esetekben így sem kerülhető el egymás üzeneteinek zavarása (pl. ha két egymást nem érzékelő csomópont megbénítja a köztük levő eszköz vételét), ezekben a rendszerekben törekednek a minél rövidebb csomagok kialakítására, így a sikeres átvitel aránya növekedhet. A csomagot ugyanis a legtöbb esetben el kell dobni, ha annak ellenőrző összege nem egyezik a feladó oldalán kiszámított és elküldött értékkel.

A TinyOS rendszerben megvalósított hálózati csomagok szerkezete látható a 6. ábrán.



6. ábra A TinyOS hálózati csomag

A *preamble* feladata, hogy korábban említett mozgó átlagot beállítsa a középvértékre, így ez a rész alternálva tartalmaz 0 és 1 biteket. A *start symbol* egy speciális bitminta, mely észlelése esetén a vevő biztos lehet benne, hogy csomag érkezik. Ezt követi a csomag fejléce a cél címmel (*addr* és *group*), a csomag típus azonosítóval (a különböző célú csomagok megkülönböztetésére az eszközön belül), majd a csomag tartalma és az ellenőrző összeg (*crc*).

Szimuláció és hibakeresés

A korábban említett konkurens működés miatt a szenzorhálózatokban történő hibakeresés meglehetősen nehézkes és bonyolult feladat. A dolgot tovább nehezíti a gazdag output perifériák hiánya, az eszközök belső állapotához nehezen lehet hozzáférni, a nyomkövetési információ kevés. Hasznos és szabványos megoldások léteznek ugyan erre a célra (pl. JTAG), mégis az egyik legfontosabb nyomkövetési lehetőség az alkalmazás szimulációja. A szimuláció ugyan sohasem egyezik teljesen a fizikai valósággal, de a belső állapotok vizsgálhatók, és a hibák determinisztikus módon újra előidézhetők.

A TinyOS rendszer része a PC-n futtatható szimulációs eszköz, a TOSSIM, mely egy diszkrét esemény alapú szimuláció, egyidőben kb. 1000 eszköz együttes működését utánozza és közvetlenül a TinyOS forráskódjából épül fel (a szimulációhoz ugyanis a TinyOS komponenseket a kereszt fordító helyett a PC hagyományos (natív) fordítójával dolgozzuk fel). A nyomkövetés alapvetően a kimentére küldött üzenetekkel valósul meg. A szimulációs környezet lehetőséget

biztosít hálózati csomagok bevitelére a szimulációs „világba” ill. az ott keletkező hálózati csomagok kinyerésére.

Szoftverkomponensek modellezése

A szimulációs eszközök – akárcsak a vizsgálandó alkalmazások – a beágyazott rendszert befogadó környezetről valamilyen modellel rendelkeznek. A modell alapú gondolkodás azonban a környezetétől elválaszthatatlan rendszer leírására ill. tervezésére is nagyon alkalmas. Sokszor – elsősorban a tervezés fázisaiban – a befogadó környezet és az alkalmazás közötti határ elmosódhat, ill. együttes modellezésük segíthet valóban „beágyazódó” alkalmazások kialakításában.

A TinyOS rendszerben megismert komponens alapú architektúra is egy modellalkotási mód a beágyazott szoftverről. Az áttekinthetőség és dokumentálás mellett további jelentősége a szoftver modelleknek az automatikus kódgenerálási lehetőség, valamint – kellően precíz és formális leírás mellett – a szoftvermodulok kompatibilitásának alapos vizsgálata. A szoftvermodell lehet egy szöveges fájl, de akár egy grafikus modellező eszközzel létrehozott komplex(hierarchikus) ábra is.

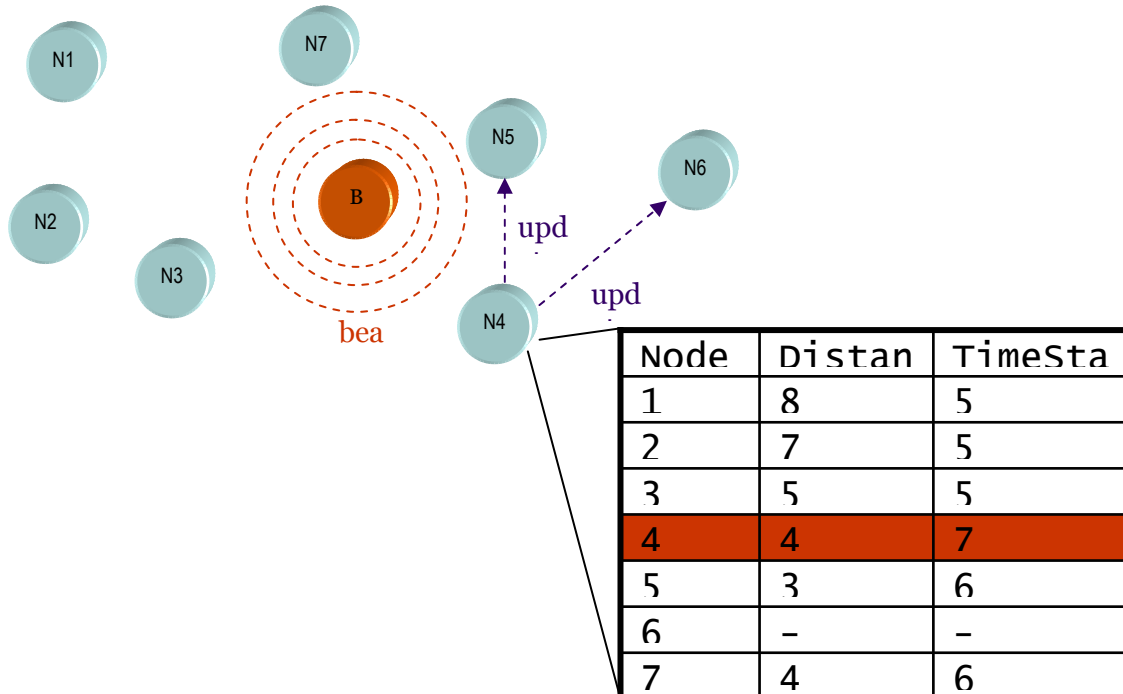
A TinyOS szoftvermodellje a komponensekbe zárt logika statikus interfészeire – a komponensekben található függvények típusaira – helyezi a hangsúlyt. Ez a szerkezeti modell lehetővé teszi alapvető fordítási időben jelentkező programozási és tervezési hibák korai felismerését ill. a komponensek típuskompatibilitásának vizsgálatát. A modell – ahogy azt látni fogjuk – megvalósítható szemléletesebb módon is, egy grafikus modellező eszköz nagyban hozzájárulhat az összetett alkalmazások szerkezetének megértéséhez. Az egyik út a szoftverkomponensek modellezéséhez tehát a meglévő koncepciók átvétele és átültetése egy (grafikus) modellező környezetbe.

A másik lehetőség, hogy jobban elszakadunk a konkrét architektúrától, nem feltétlenül vesszük át azokat a megkötéseket, amiket pl. a TinyOS komponens rendszere szab. Utóbbi megközelítésnek akkor van előnye, ha grafikus modelljeink nem csak a komponensek felületét (függvényeinek típusait) írják le, hanem a megvalósítás logikáját is definiálják. Ilyenkor teljes modelljét adjuk az alkalmazásnak, amiből egy intelligens kódgenerátor implementációs fájlokat (C fájlokat a TinyOS esetében) állít elő, és a komponensek TinyOS interfészeit is maga állítja elő. Megfelelően absztrakt leírás esetén előfordulhat, hogy a teljes alkalmazás komponensekre való partícionálásáért is a kódgenerátor a felelős. A modellezési nyelvnek ilyenkor tehát elég „erősnek” kell lennie, hogy az alkalmazás logikáját is leírassuk vele – ilyen nyelv lehet például valamilyen állapotgépes leírás. Mivel ebben az esetben a modellezési környezet jóval többet tud a szoftverről és annak belsejéről, a különböző szoftvermodulok közötti kompatibilitási vizsgálat is mélyebb lehet: nem csak fordítási hibákat ismerhetünk fel, de futási idejű inkompatibilitásra is fény derülhet.

A két modellezési szint jobb megértése végett lássunk egy-egy megvalósítást mindegyikre a TinyOS világában. Az első megközelítést a GRATIS nevű modellezési környezet követi, a második utat a DISSECT névre hallgató eszköz teszi lehetővé. Érdekességük, hogy mindkét modellezési környezet ugyanarra az általános modellezési keretrendszerre épül, mellyel a fejezet végén ismerkedünk majd meg.

Állatorvosi lovunk mindkét esetben egy nyomkövető alkalmazás lesz. A nyomkövetés ill. a követendő tárgy helyzetének meghatározása elosztottan történik a 7. ábrán látható módon. Adott területen véletlenszerűen elhelyezett érzékelők képesek a környezetükben lévő jeladó (*beacon*) érzékelésére ill. a jeladó és az érzékelő közötti távolság mérésére. Ez utóbbit egy rádiós és egy hang üzenet együttes küldésével segíti a jeladó. A kér üzenet beérkezése közötti idő alapján a távolság jól becsülhető. Minden érzékelő egy táblázatot tart fenn, melybe a saját ill. az összes többi érzékelő utolsó mért távolságát nyilvántartja. A táblázatban lévő bejegyzéseket két esetben

frissíti az érzékelő: ha saját új távolságot mért a fent ismertetett módon vagy ha valamelyik másik érzékelő elküldött hozzá más érzékelők által mért az új távolság adatokat. Ha egy érzékelő bármilyen okból frissíti a táblázatát, akkor maga is ilyen *update* üzenetet küld a szomszédjainak. Ez a logika biztosítja, hogy a hálózat minden eleme nagyjából ugyanazt a táblázatot tartalmazza és így konzisztens kép alakuljon ki a jeladó lehetséges helyzetéről mindegyik érzékelő csomóponton.

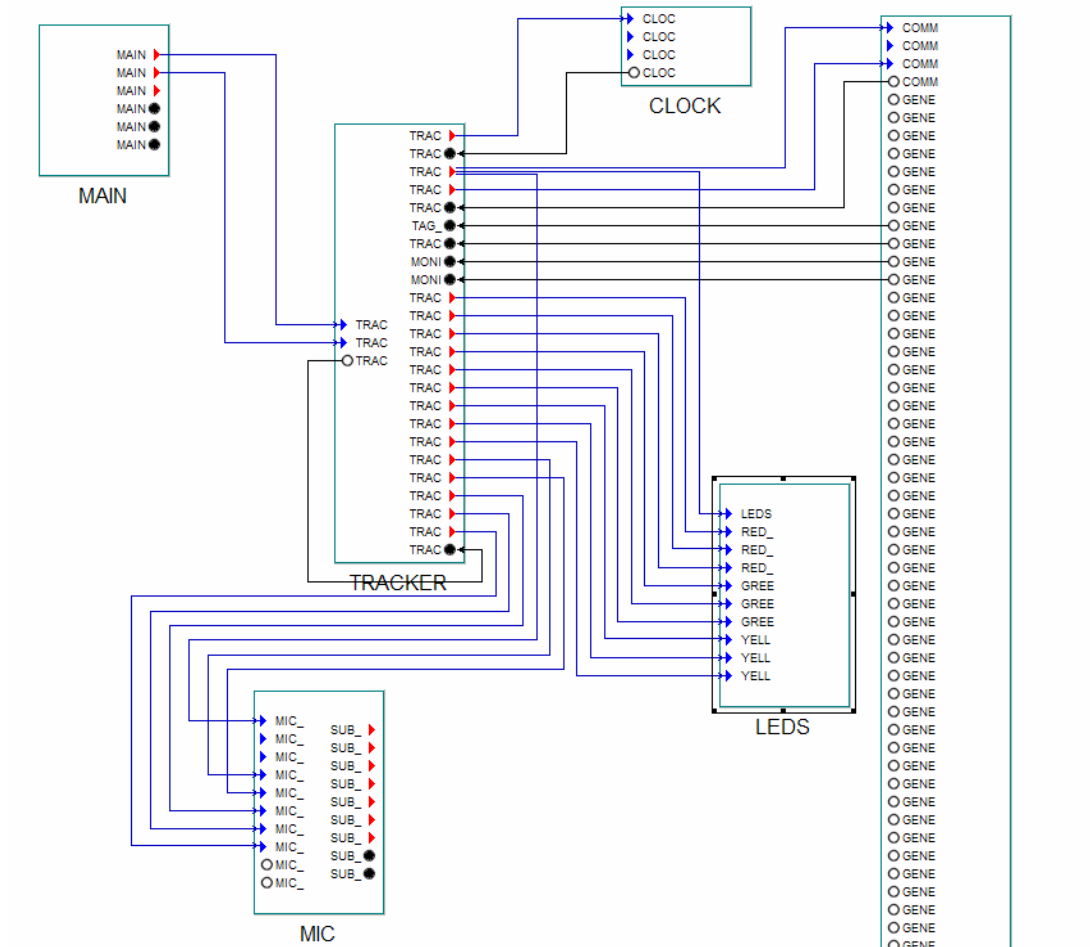


7. ábra A Tracking alkalmazás

A GRATIS nevű modellezési eszköz a Tracking alkalmazás TinyOS komponenseit segíti kialakítani. Ebben a környezetben a korábban megismert *.desc* és *.comp* fájlok sokkal szemléletesebb módon látszanak, és az eszköz ezeket a szöveges állományokat automatikusan állítja elő a grafikus modellekből. Mivel a modellezett komponensek továbbra is fekete dobozok statikus interfészekkel, továbbra is szükség van az implementáció megadására a modellezési eszközön kívül. Ehhez a GRATIS annyi segítséget ad, hogy a megírandó C fájlok vázát generálja a modellek alapján.

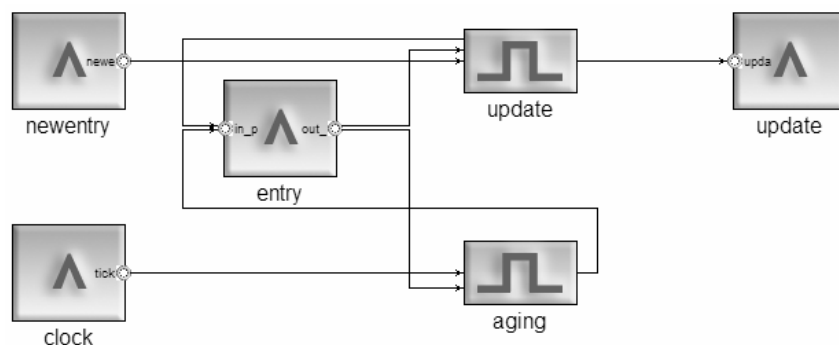
A Tracking alkalmazás legfelső szintű modelljét láthatjuk a 1. ábrán. A GRATIS modellben jól kivehetők a TinyOS komponensek: *MAIN* (inicializálás), *CLOCK* (időzítő), *LEDS* (led-ek ki- ill. bekapcsolása), *MIC* (mikrofon kezelése, mintavételezés), *GENERIC_COM* (rádiós kommunikáció). A középen látható *TRACKER* komponenst az nyomkövető alkalmazás logikáját valósítja meg. Az interfész modell alapján a hozzá tartozó C fájl vázát a GRATIS környezet automatikusan előállítja.

A DISSECT környezet tovább megy ennél. Bár nem foglalkozunk részletesen a modellezési környezet és nyelv bemutatásával, annyit érdemes megjegyezni, hogy az itt létrehozott modellekből nem kizárólag TinyOS platformra lehet kódot generálni, így e platform sajátosságai sem része ennek a nyelvnek. Alapvetően állapotok és feltételes állapotátmenetek mentén történik a modellezés. A 9. ábra szintén a Tracking alkalmazás modelljének egy részét mutatja a DISSECT környezetben.



8. ábra A Tracking alkalmazás GRATIS modellje

Az ábrán a belső táblázat karbantartásának algoritmusai látható. A háromszöget tartalmazó dobozok a bemeneti adatok ill. a belső állapotok (memória) egy konfigurációját írják le, míg az órajellel jelölt elemek a feldolgozást (állapotátmenetet és adattranszformációt) írják le. Az ábra jelentése nagy vonalakban ennyi: *newentry* input érkezése esetén az *update* lépés egyrészt beírja az adatot a memóriába, ha az valóban új adat, másrészt tovább küldi (a rádiós komponens felé). Az időzítő hatására, mely egy egyszerű *clock* tokenet generál, az *aging* átmenet a táblázat bejegyzéseit „öregbíti”, így nagyon régi méréseket egy idő után majd eldobhatunk.

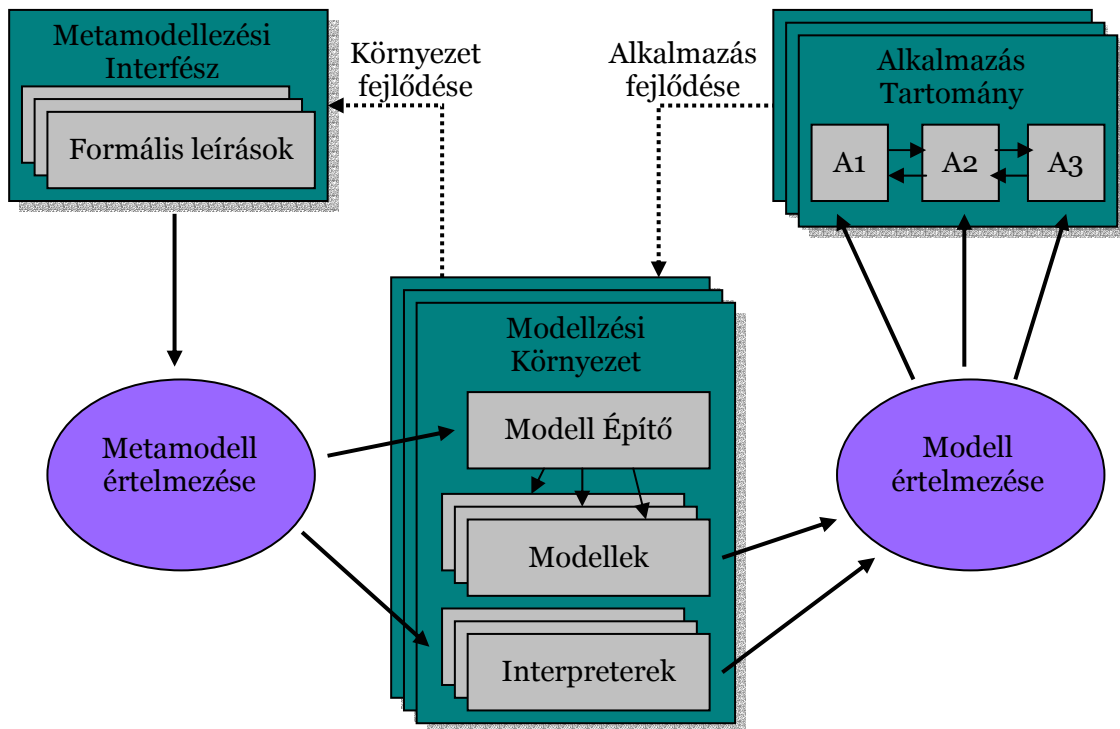


9. ábra A Tracking alkalmazás modellje a DISSECT környezetben

Ahogy láthattuk, a modellezés során különböző absztrakciós szinteket választhatunk a szoftver tervezéséhez (esetleg megvalósításához). Adott szinten is azonban alkalmazás közben változhatnak a koncepciók, a modellezési nyelv szabályai. Ha a modellezési eszközünket magunk készítjük el a célnak megfelelően, könnyen lehet, hogy egy apró nyelvi változtatás temérdek

munkát jelent a modellezési eszköz mosósításánál. Ennek elkerülése miatt hoztak létre különböző általános modellezési eszközöket, melyeket rugalmasan konfigurálhatunk konkrét modellezési nyelvekhez (*domain specific language*). A GRATIS és a DISSECT ugyanazt az általános modellezési keretrendszert használja: a Vanderbilt Egyetemen kifejlesztett GME (*Generic Modeling Environment*) eszközt.

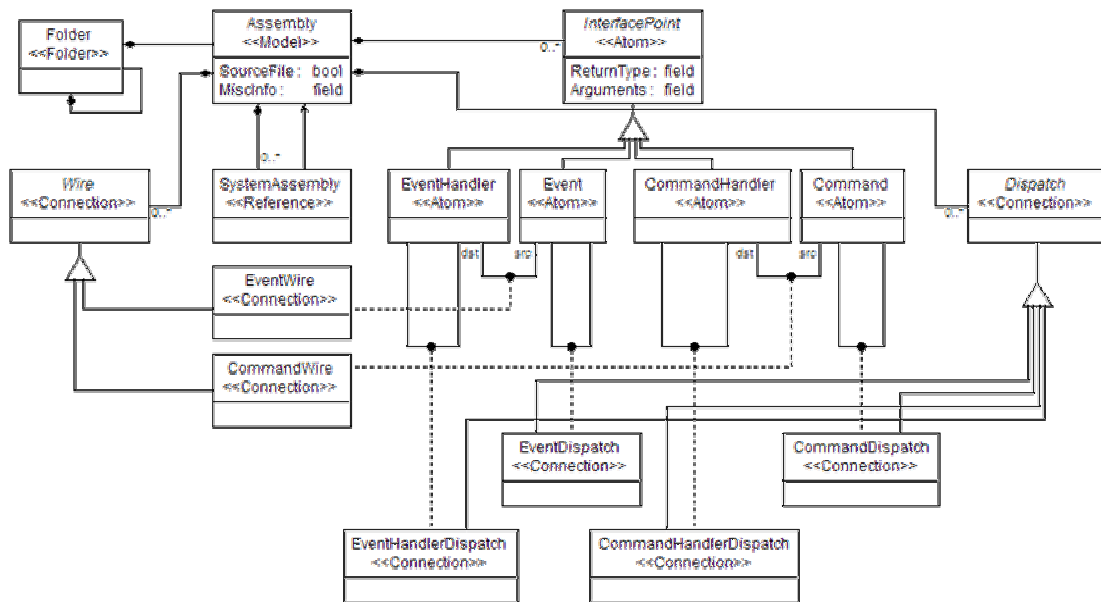
A GME modellezési környezet ún. *metamodell*ek segítségével konfigurálható a kívánt modellezési nyelv kezelésére. A metamodell tartalmazza a modellezési nyelvünk (pl. GRATIS) szereplőit (TinyOS komponensek), azok közötti lehetséges kapcsolatokat (huzalozás), speciális szabályokat és a megjelenítés mikéntjét (pl. ikon alakja). A metamodell használata azonban nem a GME eszközre jellemző egyedi vonás: a relációs adatbázis-kezelők tábladefiníciói is ilyen metamodell, melyekkel egy általános adatbázis-kezelő megtanítható bizonyos szabályokra. A GME környezet érdekessége, hogy a metamodell ugyanazzal az eszközzel készülnek, mint a későbbi valódi modellek. A GME keretrendszer egyetlen „beépített” modellezési környezete tehát a metamodellezési környezet. A metamodell tulajdonképpen a kialakítandó nyelv szintaktikáját írja le, a későbbi modellek értelmezése (szemantika) a GME keretrendszerhez illeszthető ún. interpreterek segítségével/implementálásával oldható meg.



10. ábra Egy rugalmas modellezési keretrendszer

A 10. ábra egy rugalmas modellezési keretrendszerben történő fejlesztés menetét mutatja. Az ábra bal oldalán látható kör a modellezési nyelv, a modellezési koncepciók iteratív fejlesztését, javítását teszi lehetővé párhuzamosan a jobb oldalon látható alkalmazás (konkrét modell) fejlesztésével. Ez a megközelítés arra a nagyon fontos felismerésre épít, hogy a modellezési nyelv csiszolásához szükség van konkrét modellek építésére, a tanulságok felhasználására. Mindkét körfolyamat hatással van tehát a másikra.

Végezetül lássuk a GRATIS modellezés környezet metamodelljét (egy konkrét modellt már láthattunk korábban a nyomkövető alkalmazás ismertetésénél). A GME metamodelljei nagyon sokban hasonlítanak a UML nyelv osztálydiagramjaihoz.



11. ábra A GRATIS környezet metamodellje

Az metamodelen érdemes megkeresni az esemény és parancskezelő entitásokat (*EventHandler*, *CommandHandler*, *Event*, *Command*). Ezeket tartalmazhatja egy *Assembly* objektum (mely a TinyOS komponst fogja megtestesíteni a nyelvben), és melyek között *Wire* nevű kapcsolat húzható.

A DISSECT környezet is rendelkezik egy fentihez hasonló metamodellel, azonban ez jóval bonyolultabb. Mivel ez a modellezési nyelv meglehetősen sajátos, nem foglalkozunk vele a továbbiakban. Léteznek azonban olyan logikát is leíró nyelvek, melyek szabványosnak (elfogadottnak, kipróbáltak) tekinthetők. Ezeket számítási modelleknek nevezzük, és a beágyazott rendszerek modellezése során nagy jelentőséggel bírnak. A következő részben ezekkel a nyelvekkel ismerkedünk meg részletesen.