10 FORMÁLIS SPECIFIKÁCIÓ

TÉMAKÖRÖK

A fejezet célja, hogy bemutasson olyan formális specifikációs technikákat, amelyeket fel lehet használni a rendszerkövetelmény-specifikáció részletekkel történő bővítéséhez. A fejezet elolvasása után:

- meg fogjuk érteni, miért segítik a formális specifikációs technikák a rendszerkövetelményekben lévő problémák felderítését;
- meg fogjuk érteni a formális specifikáció algebrai technikáinak használatát az interfész-specifikációk meghatározásához;
- meg fogjuk érteni, hogyan használhatók a formális, modellalapú technikák a viselkedés specifikációjára.

TARTALOM

A „hagyományos" mérnöki tudományokban, mint amilyen a villamosmérnöké vagy az építészmérnöké, a fejlődés rendszerint jobb matematikai technikák kifejlesztésével is járt. A mérnöki iparnak nem okozott nehézséget, hogy elfogadja a matematikai elemzés szükségességét, és azt beépítse a folyamataiba. A matematikai elemzés a terméktervek fejlesztésének és validálásának megszokott részét képezi.

A szoftvertervezés ellenben nem ezt az utat követte. Bár ma már több mint 25 év kutatás fekszik a szoftverfolyamat matematikai technikáinak használatában, ezek a technikák igen korlátozott hatást gyakoroltak. A szoftverfejlesztés úgynevezett formális módszereit nem használják széles körben az ipari szoftverfejlesztésben. A legtöbb szoftverfejlesztő cég nem tekinti ezeket elég költséghatékonynak ahhoz, hogy alkalmazza azt szoftverfejlesztési folyamataiban.

Formális módszer elnevezéssel illetünk minden olyan tevékenységet, amely a szoftver matematikai reprezentációira támaszkodik, köztük a formális rendszerspecifikációt, a specifikációelemzést és -bizonyítást, a transzformációs fejlesztést és a programverifikációt. Ezen összes tevékenység alapja a szoftver formális spe-

cifikációja. A formális specifikáció egy olyan nyelven kifejezett specifikációt jelent, amelynek szókészletét, szintaktikáját és szemantikáját formális módon definiálták. A formális definíciónál ez a követelmény azt jelenti, hogy a specifikációs nyelvnek olyan matematikai fogalmakon kell alapulnia, amelyek tulajdonságai kellően tiszták. A használt matematikai ágat diszkrét matematikának hívják, a matematikai fogalmak pedig a halmazelméletből, a logikából és az algebrából kerültek át.

A 80-as években sok szoftverfejlesztéssel foglalkozó kutató a formális fejlesztési módszerek használatát tekintette a szoftverminőség javítása legjobb módjának. Azzal érveltek, hogy a formális módszerek lényeges részét képező szigorú és részletes elemzés olyan programokhoz vezet majd, amelyek kevesebb hibát tartalmaznak és jobban illeszkednek a felhasználók igényeihez. Azt jósolták, hogy a 21. századra a szoftverek nagy hányadát formális módszerek segítségével fejlesztik majd.

Látható, hogy ez a jóslat nem vált valóra. Ennek négy fő oka van:

1. *Sikeres szoftvertervezés.* Az olyan egyéb szoftvertervezési módszerek használata a szoftvertervezési és -fejlesztési folyamatban, mint a strukturált módszerek, a konfigurációkezelés, az információrejtés, javulást eredményeztek a szoftverek minőségében. Világosan látszik, hogy nem volt igazuk azoknak, akik szerint a szoftverminőség javításának egyedüli módja a formális módszerek használata.
2. *Piaci változások.* A 80-as években a szoftverek minőségét látták a szoftvertervezés kulcsfontosságú problémájának. Azóta viszont a szoftverfejlesztés számos osztályánál nem a minőség a kritikus kérdés, hanem a piacra kerülési idő. A szoftvereket gyorsan kell fejleszteni, és a megrendelők gyakran hibákkal együtt is hajlandók elfogadni a szoftvert, ha ezzel elérhető a gyorsabb szállítás. A gyors szoftverfejlesztési technikák nem működnek hatékonyan együtt a formális specifikációkkal. A minőség természetesen még mindig fontos tényező, de a gyors szállítással összefüggésben kell elérni azt.
3. *A formális módszerek korlátozott körű alkalmazhatósága.* A formális módszerek nem alkalmasak felhasználói interfészek és felhasználói interakciók specifikálására. A felhasználói interfész komponens a legtöbb rendszer egyre nagyobb részét alkotja, így formális módszereket valójában csak a rendszer többi részének fejlesztésében lehet használni.
4. *A formális módszerek korlátozott skálázhatósága.* A formális módszerek még mindig nem skálázhatók jól. Az ilyen technikákat igénybe vevő sikeres projektek legfőképp viszonylag kis, kritikus rendszermagokkal foglalkoznak. Ahogy a rendszerek mérete növekszik, úgy a formális specifikáció kifejlesztéséhez szükséges idő és erőfeszítés is aránytalanul megnő.

Ezek a tényezők azt jelentik, hogy a formális módszerek alkalmazásának kockázata a legtöbb szoftverprojektnél magasabb a használatukból származó lehetséges előnyöknél. A formális módszerek bevezetése a szoftverfolyamatoknál magas költségeket és sok problémát okoz. Ugyanakkor a formális specifi-

káció kitűnő módot kínál arra, hogy hibákat fedjünk fel a specifikációban, és a rendszer-specifikációt egyértelmű módon mutassuk be. Minden olyan projekt, amely formális módszereket használt, kevesebb hibáról számolt be az átadott szoftverben.

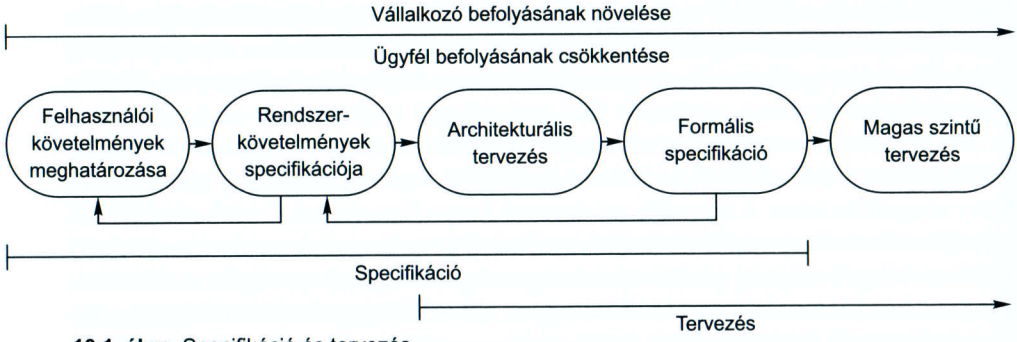
Emiatt azokban a rendszerekben, ahol el kell kerülni a meghibásodásokat, mérlegelhetjük a formális módszerek használatát, és az valószínűleg költséghatékony megoldás lehet. A formális módszerek használata terjed a kritikus rendszerek fejlesztésének specializált területén, ahol nagyon fontosak az olyan eredendő rendszertulajdonságok, mint a biztonságosság, a megbízhatóság és a védelem. Ezeknek a kritikus rendszereknek, ahogyan azt a 24. fejezetben látni fogjuk, igen magasak a validálási költségei, és a rendszer meghibásodásának költségei is nagyok és egyre növekvőek. Azért használnak formális módszereket, mert azok csökkenteni tudják ezeket a költségeket.

Azok között a kritikus rendszerek között, melyeknél a formális módszereket sikerrel alkalmazták, szerepel egy légiforgalom-irányítási ellenőrző rendszer (Hall, 1996), egy vasúti jelzőrendszer (Dehbonei és Mejia, 1995), űrrepülési rendszerek (Easterbrook *és mások*, 1998) és orvosi vezérlőrendszerek (Jacky, 1995; Jacky *és mások*, 1997). A formális módszereket szintén használták szoftvereszközök specifikációjára (Neil *és mások*, 1998), az IBM CICS-rendszere egy részének specifikációjára (Wordsworth, 1991) és egy valós idejű rendszermaghoz (Spivey, 1990). A szoftverfejlesztés Cleanroom-módszere (Mills *és mások*, 1987; Linger, 1994; Prowell *és mások*, 1999), formálisan megalapozott érvelésre támaszkodik, amely kód összhangban van annak specifikációjával. Az Egyesült Királyságban a védelmi minisztérium előírja a formális módszerek használatát biztonságosságkritikus rendszereknél (MOD, 1995).

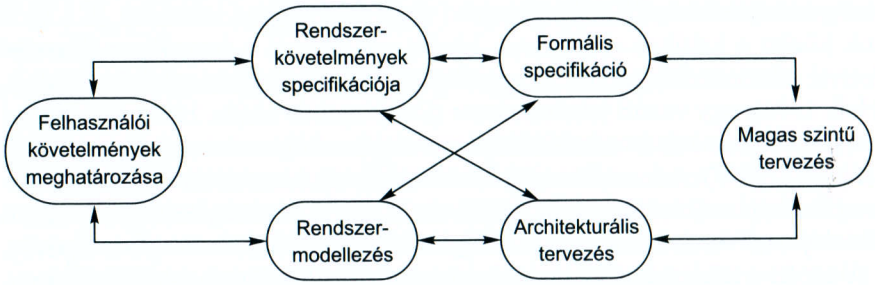
10.1. FORMÁLIS SPECIFIKÁCIÓ A SZOFTVERFOLYAMATBAN

A kritikus rendszerek fejlesztése általában magában foglal egy tervalapú szoftverfolyamatot, amely a 4. fejezetben tárgyalt vízesésmodellű fejlesztésen alapul. A rendszerkövetelmények és a rendszerterv egyaránt részletesen ki van fejtve, és az implementáció kezdete előtt gondosan elemezték és ellenőrizték. Ha elkészítik egy szoftver formális specifikációját, azt általában a követelmények specifikációja után, de még a részletes rendszerterv előtt teszik. Szoros visszacsatolási kör van a részletes követelményspecifikáció és a formális specifikáció között. Amint azt később látni fogjuk, a formális specifikáció előnye, hogy képes felderíteni a rendszerkövetelményekkel kapcsolatos problémákat és kétértelműségeket.

A specifikáció kifejlesztésével visszaszorul az ügyfél bevonása a szoftverfejlesztésbe, a vállalkozóé pedig nő, ahogy a rendszer-specifikáció több részlettel bővül. A folyamat korai szakaszában a specifikációnak ajánlott megrendelő-központúnak lennie. Érdemes azt az ügyfél számára érthető módon megírnunk, és lehetőleg olyan keveset tételezzünk fel a szoftvertervről, amilyen keveset csak



10.1. ábra. Specifikáció és tervezés



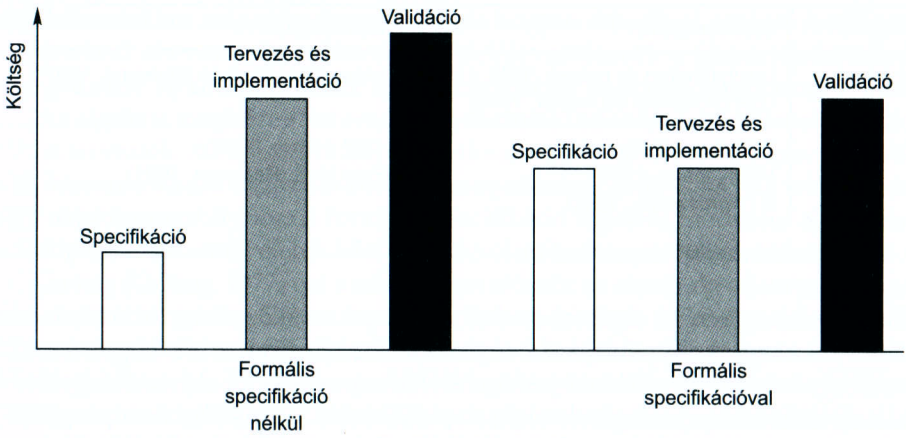
10.2. ábra. Formális specifikáció a szoftverfolyamatban

lehet. Ugyanakkor a folyamat utolsó szakasza, amely egy teljes, ellentmondásmentes és pontos specifikáció létrehozásából áll, elsősorban a szoftver szállítójának szól. Ez írja le a rendszer-implementáció részleteit. Ebben a szakaszban formális nyelvet használhatunk ahhoz, hogy elkerüljük a többértelműségeket a szoftverspecifikációban.

A 10.1. ábra bemutatja a szoftverspecifikáció szakaszait és annak kapcsolódását a tervezési folyamathoz. A 10.1. ábrán bemutatott specifikációs szakaszok nem függetlenek, és nem is kell azokat az ott látható sorrendben kifejleszteni. A 10.2. ábrán az látható, hogy a specifikációs és tervezési tevékenységek egymással párhuzamosan is végezhetők. A folyamatban minden szakasz között kétirányú kapcsolat van. A specifikáció ellátja információval a tervezési folyamatot, és fordítva.

Ahogy fejlesztjük a részletes specifikációt, úgy egyre jobban megértjük azt. Formális specifikáció létrehozása részletes rendszerelemzést követel tőlünk, amely rendszerint hibákat és ellentmondásosságot tár fel az informális követelményspecifikáción belül. Ez a hibafelismerés valószínűleg a legmeggyőzőbb érv a formális specifikáció kifejlesztése mellett (Hall, 1990). Segít felderíteni azokat a követelménybeli problémákat, amelyek kijavítása később nagyon költséges lehet.

A használt folyamattól függően, a formális elemzés során felderített specifikációbeli problémák változtatásokat eredményezhetnek a követelményspecifikációban, ha arról még nem egyeztek meg. Ha a követelményspecifikációt elfogadták, és belevették a rendszerfejlesztésről kötött szerződésbe, akkor a megrendelővel



10.3. ábra. Szoftverfejlesztési költségek alakulása formális specifikáció használatánál

együtt felfedezett problémákat ajánlatos kiemelni. Utána már a megrendelőn múlik, hogy eldöntse, hogyan lehetne azokat megszüntetni a rendszer tervezésének megkezdése előtt.

Egy formális specifikáció kifejlesztése és elemzése átrendezi a szoftverfejlesztés költségeit. A 10.3. ábra megmutatja, hogy valószínűleg hogyan érinti a szoftverfolyamat költségeit a formális specifikáció használata. Hagyományos folyamat használatánál a validálási költségek a fejlesztési költségek mintegy 50 százalékát teszik ki, az implementálás és tervezés költsége pedig kb. kétszerese a specifikáció költségének. Formális specifikációval a specifikáció és az implementáció költsége közel egyenlő, a rendszer validálásának költsége viszont jelentősen csökken. Mivel a formális specifikáció kifejlesztése követelményi problémákat tár fel, elkerülhető, hogy a rendszer megtervezése után újra kelljen írni valamit ezeknek a problémáknak a kijavítására.

Két formális specifikációs megközelítést használtak már ipari szoftverrendszerek részletes specifikációjának megírására. Ezek a következők:

1. Az *algebrai megközelítés,* ahol a rendszer a műveletek és azok kapcsolatai szempontjából van leírva.
2. A *modellalapú megközelítés,* ahol a rendszer modellje olyan matematikai konstrukciók segítségével van felépítve, mint a halmazok és a sorozatok, a rendszer műveletei pedig aszerint vannak definiálva, hogyan módosítják a rendszer állapotát.

Ezen családokon belül különböző nyelveket fejlesztettek ki, szekvenciális és konkurens rendszerek specifikálására. A 10.4. ábrán mindkét osztályba tartozó nyelvekre láthatunk példát. A táblázatban láthatjuk, hogy e nyelvek közül a legtöbbet a 80-as években fejlesztették ki. Egy formális specifikációs nyelv finomítása éveket vesz igénybe, így a legtöbb formális specifikációs kutatás ezeken a nyelveken alapszik, és nem foglalkozik új jelölések kitalálásával.

	Szekvenciális	Konkurens
Algebrai	Larch (Guttag *és mások*, 1985, 1993) OBJ (Futatsugi *és mások*, 1985)	Lotos (Bolognesi és Brinksma, 1987)
Modellalapú	Z (Spivey, 1992) VDM (Jones, 1980) B (Wordsworth, 1996)	CSP (Hoare, 1985) Petri Nets (Peterson, 1981)

10.4. ábra. Formális specifikációs nyelvek

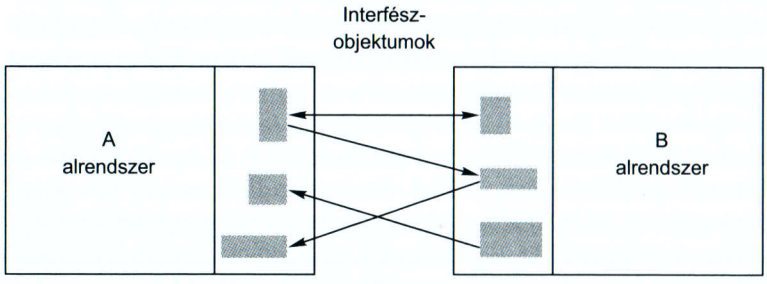
Ebben a fejezetben az algebrai és a modellalapú megközelítést is be kívánom mutatni. Az itt szereplő példákkal benyomást szerezhetünk arról, hogyan eredményez egy formális specifikáció precíz, részletes specifikációt, de nem tárgyalom majd a specifikációs nyelvi részleteket, a specifikációs technikákat és a programverifikáció részleteit. A könyv weboldaláról mindkét technikáról elérhető egy részletesebb leírás.

10.2. ALRENDSZERINTERFÉSZEK SPECIFIKÁCIÓJA

A nagyméretű rendszereket általában alrendszerekre bontják szét, amelyeket egymástól függetlenül fejlesztenek. Az alrendszerek használhatnak más alrendszereket, így a specifikációs folyamatnak lényeges része az alrendszerinterfészek meghatározása. Mikor az interfészekről megegyeznek és definiálják azokat, az alrendszerek egymástól függetlenül megtervezhetők és implementálhatók.

Az alrendszerek interfészeit gyakran definiálják objektumok vagy komponensek halmazaként (10.5. ábra). Ezek leírják azokat az adatokat és műveleteket, amik az alrendszer interfészén keresztül elérhetők. Az alrendszerinterfész-specifikációt tehát előállíthatjuk az interfészt alkotó objektumok specifikációinak kombinálásával.

A pontos alrendszerinterfész-specifikációk azért fontosak, mert az alrendszerek fejlesztőinek azelőtt kell megírniuk a többi alrendszer szolgáltatásait használó kódot, mielőtt azokat implementálnák. Az interfész-specifikáció információt nyújt az alrendszerfejlesztőknek ahhoz, hogy tudják, milyen szolgáltatások állnak majd



10.5. ábra. Alrendszerinterfész-objektumok

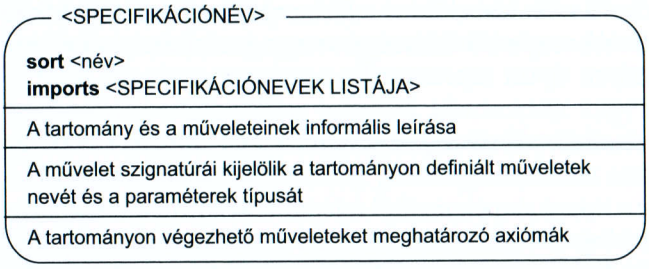
rendelkezésükre más alrendszerekben, és hogyan érhetik el azokat. A világos és egyértelmű alrendszerinterfész-specifikációk csökkentik a félreértések esélyét a szolgáltatást nyújtó alrendszer és a szolgáltatást használó alrendszerek között.

Az algebrai megközelítést eredetileg absztrakt adattípus-interfészek definíciójához tervezték. Az absztrakt adattípusoknál a típust nem annak reprezentációjával, hanem a hozzá tartozó műveletek megadásával definiálják. Ez tehát hasonló egy objektumosztályhoz. A formális specifikáció algebrai módszere az absztrakt adattípust a típusműveletek közötti kapcsolatok szempontjából definiálja.

Guttag (Guttag, 1977) ezt a szemléletet először az absztrakt adattípusok specifikációjánál tárgyalta. Cohen és mások (Cohen *és mások*, 1986) egy dokumentumvisszakereső rendszer példájának segítségével megmutatják, hogyan terjeszthető ki a technika teljes rendszer-specifikációig. Liskov és Guttag (Liskov és Guttag, 1986) szintén foglalkozik az absztrakt adattípusok algebrai specifikációjával.

A 10.6. ábrán egy objektumspecifikáció szerkezete látható. A specifikáció törzse négy komponensből áll:

1. *Bevezetésből*, amely a specifikált egyed tartományát (a típusnevet) deklarálja. A *tartomány* közös jellemzővel bíró objektumok egy halmazának a neve. Hasonló a programozási nyelvek típusához. A bevezetés szintén tartalmazhat egy „imports" deklarációt, melyben más tartománydefiniáló specifikációk nevei vannak deklarálva. Egy specifikáció importálása lehetővé teszi ezeknek a tartományoknak a használatát.
2. *Leírás* részből, ahol a műveleteket informálisan írják le. Ez a formális specifikációt könnyebben érthetővé teszi. A formális specifikáció ezt a leírást azzal egészíti ki, hogy egyértelmű szintaktikát és szemantikát biztosít a típusműveletekhez.
3. A *szignatúra* rész az interfész szintaktikáját határozza meg az objektumosztályhoz vagy az absztrakt adattípushoz. A szignatúrában a definiált műveletek neve, azok paramétereinek neve és tartománya, valamint a művelet eredményének tartománya van leírva.
4. Az *axióma* rész a műveletek szemantikáját egy axiómahalmaz megadásával definiálja, amely az absztrakt adattípusok viselkedését jellemzi. Ezek az axiómák a definiált tartomány egyedeinek létrehozására használt műveleteket olyan műveletekkel kapcsolják össze, melyekkel azok értékei vizsgálhatók.



10.6. ábra. Algebrai specifikáció struktúrája

Egy alrendszerinterfész formális specifikációjánál a fejlesztési folyamat a következő tevékenységeket tartalmazza:

1. *Specifikáció strukturálása.* Szervezzük az informális interfész-specifikációt absztrakt adattípusok vagy objektumosztályok egy halmazába. Ajánlott informálisan meghatároznunk minden egyes osztályhoz a hozzá kapcsolódó műveleteket.
2. *Specifikáció elnevezése.* Határozzunk meg egy nevet minden absztrakt típus specifikációjához, döntsük el, hogy szükségük van-e azoknak generikus paraméterekre, és döntsünk az azonosított tartományok neveiről.
3. *Műveletek kiválasztása.* Válasszuk ki műveletek egy csoportját minden specifikációhoz, az azonosított interfész funkciói alapján. Ajánlott műveleteknek szerepelnie a tartomány példányainak létrehozásához, a példányok értékének módosításához és a példányok értékének vizsgálatához. Lehet, hogy az informális interfész-definícióban eredetileg azonosított funkciókat újakkal kell kiegészíteni.
4. *Informális művelet specifikációja.* Írjunk informális specifikációt minden művelethez. Le kell írnunk, milyen hatással vannak a műveletek a definiált tartományra.
5. *Szintaktikadefiníció.* Definiáljuk a műveletek szintaktikáját és a paramétereket minden művelethez. Ez a formális specifikáció szignatúra része. Frissítsük az informális specifikációt ebben a szakaszban, amennyiben az szükséges.
6. *Axiómadefiníció.* Definiáljuk a műveletek szemantikáját annak leírásával, milyen feltételek igazak mindig a műveletek különböző kombinációjánál.

Az algebrai specifikáció technikájának elmagyarázására egy egyszerű adatszerkezet (láncolt lista) példáját használom. Ezt szemlélteti a 10.7. ábra. A láncolt listák rendezett adatszerkezetek, amelyben minden elem tartalmaz hivatkozást az adatszerkezet következő elemére. Egy egyszerű, néhány műveletes listát használok, hogy a tárgyalás ne legyen túl hosszú. A gyakorlatban a listát definiáló osztályok rendszerint több művelettel rendelkeznek.

Tegyük fel, hogy a specifikációs folyamat első szakaszán, a specifikációk rendszerezésén már túl vagyunk, és felismertük, hogy egy listára van szükségünk. A specifikáció és a tartomány neve ugyanaz is lehet, bár hasznos ezeket valamilyen konvenció szerint megkülönböztetni. Itt nagybetűkkel jelöltem a specifikáció nevét (LISTA), és nagy kezdőbetű után kisbetűkkel a tartomány nevét (Lista). Mivel a listák más típusok kollekciói, a specifikációnak van egy generikus paramétere (Elem). Az Elem név bármilyen típust reprezentálhat: egészet, sztringet, listát és így tovább.

Általánosságban minden absztrakt adattípusra igaz, hogy a kívánt műveletek között szerepelnie kell olyan műveletnek, amely példányokat hoz létre a típushoz (Create), illetve amely a típust alapelemeiből felépíti (Cons). Listák esetében ajánlott egy olyan művelet létezése, amely hozzáfér az első elemhez (Head), egy olyannak, amely visszatér az első elem eltávolításával nyert listával (Tail), valamint egy olyannak, amely a lista elemeinek számát adja vissza (Length).

LISTA(Elem)

sort Lista
imports INTEGER

Olyan listát definiál, ahol az új elem a lista végére kerül és az első elemet lehet eltávolítani.
A műveletek: Create, amely üres listát hoz létre; a Cons, amely új listát hoz létre, kiegészítve az adott taggal, a Length, amely megadja a lista méretét, a Head, amely megadja a lista első elemét és a Tail, mely létrehoz egy listát a bemeneti listából a fej eltávolításával.
Az Undefined az Elem típus nem definiált értékét reprezentálja.

Create → Lista
Cons (Lista, Elem) → Lista
Head (Lista) → Elem
Length (Lista) → Integer
Tail (Lista) → Lista

Head (Create) = Undefined **exception**(üres lista)
Head (Cons (L, v)) = **if** L = Create **then** v **else** Head (L)
Length (Create) = 0
Length (Cons (L, v)) = Length (L) + 1
Tail (Create) = Create
Tail (Cons (L, v)) = **if** L = Create **then** Create **else** Cons (Tail (L), v)

10.7. ábra. Egyszerű listaspecifikáció

Ahhoz, hogy megadjuk ezen összes művelet szintaktikáját, el kell döntenünk, milyen paraméterek szükségesek a művelethez és milyen annak visszatérési típusa. Általánosságban bemenő paraméter az éppen definiált tartomány (Lista) vagy a generikus tartomány (Elem) lehet. A műveletek eredményei vagy ugyanazokhoz a tartományokhoz tartoznak, vagy valamely más tartományhoz, mint az Integer vagy a Boolean. A lista példában a Length művelet egy egészet ad vissza. Ezért a specifikációban szerepeltetnünk kell egy „imports" deklarációt arról, hogy a specifikációban használjuk az integer specifikációját.

A specifikáció létrehozásához definiálunk egy axiómahalmazt, amely teljesül az absztrakt adattípusra, és leírja annak szemantikáját. Az axiómákat a szignatúra részben definiált műveletekkel írjuk le. Ezek a szemantikát annak kijelölésével írják le, hogy mely állítások igazak mindig az egyedek viselkedésével kapcsolatban az absztrakt adattípusnál.

Az absztrakt adattípuson végzett műveletek rendszerint két csoportra bomlanak:

1. *Konstruktorműveletekre,* amelyek létrehozzák vagy módosítják a specifikációban definiált tartomány egyedeit. Ezek jellemzően olyan neveket kapnak, mint a Create, az Update, az Add vagy ebben az esetben a Cons, amely konstrukciót jelent.
2. *Lekérdezőműveletekre,* amelyek lekérdezik a specifikációban definiált tartomány attribútumait. Ezek jellemzően olyan neveket kapnak, mint az Eval vagy a Get.

Az algebrai specifikáció írásának hasznos ökölszabálya, hogy adjuk meg a konstruktorműveleteket, és minden lekérdezőművelethez minden konstruktor mellett írjunk egy axiómát. Eszerint m konstruktorművelet és n lekérdezőművelet esetén m × n axiómát ajánlott definiálnunk.

Ugyanakkor nem biztos, hogy egy absztrakt típus konstruktorműveleteinek mindegyike primitív konstruktor. Vagyis ezek más konstruktorok és lekérdezőműveletek segítségével is definiálhatók. Ha egy konstruktorműveletet más konstruktorok segítségével definiálunk, akkor csak a primitív konstruktorokat használó lekérdezőműveleteket szükséges megadnunk.

A lista specifikációjában a listát létrehozó konstruktorműveletek a Create, a Cons és a Tail. A lekérdezőműveletek a Head (a lista első elemét adja vissza) és a Length (a lista elemeinek számát adja vissza), melyek a lista attribútumainak felderítésére használhatók. A Tail művelet, mindamellett nem primitív konstruktor. Emiatt nincsen szükség arra, hogy axiómákat definiáljunk a Head és Length műveletekhez a Tail művelet mellett, de a Tail-t definiálnunk kell a primitív konstruktorműveletek segítségével.

Egy üres lista fejének kiértékelése definiálatlan értéket eredményez. A Head és a Tail specifikációja azt mondja, hogy a Head a lista első elemét adja értékül, a Tail pedig a bemenő listát, annak fejét elhagyva. A Head specifikációjában az áll, hogy a Cons segítségével létrehozott lista feje vagy a listához adott érték (ha az eredeti lista üres), vagy megegyezik a Cons-nak kezdetben átadott lista paraméter fejével. Egy elem hozzáadása a listához nem érinti a lista fejét, hacsak nem üres listáról van szó.

A rekurzió használata teljesen megszokott az algebrai specifikációk írásánál. A Tail művelet értéke az a lista, amely a bemenő listából a feje elhagyásával alakul ki. A Tail definíciója megmutatja, hogyan használják a rekurziót az algebrai specifikációk létrehozásánál. A művelet definiálva van üres listákra, azután pedig rekurzív módon a nem üres listákra is úgy, hogy a rekurzió akkor áll le, ha üres lista tér vissza.

Néha könnyebb megérteni a rekurzív specifikációkat egy rövid példa segítségével. Mondjuk, hogy van egy [5, 7] listánk, ahol az 5 a lista eleje, a 7 pedig a lista vége. A Cons ([5, 7], 9) műveletnek az [5, 7, 9] listával kell visszatérnie, az erre alkalmazott Tail műveletnek pedig a [7, 9] listával. Az egyenlőségek sorozata, amelyet úgy kapunk, hogy a fenti specifikációban a paramétereket ezekkel az értékekkel helyettesítünk, a következő:

```
Tail ([5, 7, 9]) =
  Tail (Cons ([5, 7], 9)) =
  Cons (Tail ([5, 7]), 9) =
  Cons (Tail (Cons ([5], 7)), 9) =
  Cons (Cons (Tail ([5]), 7), 9) =
    Cons (Cons (Tail (Cons ([], 5)), 7), 9) =
    Cons (Cons ([Create], 7), 9) =
  Cons ([7], 9) =
    [7, 9]
```

A példából látható, hogy a Tail axiómájának módszeres beírása valóban a várt eredményt hozza. Ugyanezt az átírásos technikát használva ellenőrizhetjük, hogy a Head axiómája is helyes.

Most nézzük meg, hogyan használhatjuk egy interfész algebrai specifikációját kritikus rendszereknél. Tegyük fel, hogy egy légiforgalom-irányítási ellenőrző rendszernél egy objektumot egy ellenőrzött légi szektor reprezentálására terveztek. Minden ellenőrzött szektorban több repülőgép is lehet, amelyek mind egyedi repülőgép-azonosítóval rendelkeznek. Biztonsági okokból a repülőgépek között legalább 300 méteres magasságkülönbségnek kell lennie. A rendszer figyelmezteti az irányítót, ha egy pilóta megkísérli a gépét olyan pozícióba vinni, amivel megsérti ezt a megszorítást.

A leírás egyszerűsítése érdekében a szektor objektumhoz csupán korlátozott számú műveletet definiáltunk. Egy valós rendszerben valószínűleg sokkal több művelet létezik, a repülőgépek horizontális elkülönítésére pedig összetettebb biztonsági feltételek vonatkoznak. Az objektumon a kritikus műveletek a következők:

1. Enter Ez a művelet egy repülőgépet (amelyet az azonosítójával reprezentálunk) hozzáad a légtérhez a megadott magasságban. Más repülőgép nem lehet abban a magasságban vagy annak 300 méteres közelségében.
2. Leave Ez a művelet a megadott repülőgépet eltávolítja az ellenőrzött szektorból. Ezt a műveletet használják, ha a repülőgép egy szomszédos szektorba kerül.
3. Move Ez a művelet egy repülőgépet bizonyos magasságról egy másik magasságra mozgat. A repülőgépek 300 méteres függőleges elkülönítésére vonatkozó biztonsági megszorítása itt is ellenőrzésre kerül.
4. Lookup A repülőgép azonosítóját megadva a művelet visszaadja a repülőgép aktuális magasságát a szektorban.

Egyszerűbb specifikálni ezeket a műveleteket, ha bizonyos egyéb interfészműveletek is definiálva vannak. Ezek a következők:

1. Create Ez szokványos művelet absztrakt adattípusoknál. Ez a típus egy üres példányát hozza létre. Ebben az esetben ez olyan szektort reprezentál, amelyben nincsen repülőgép.
2. Put Ez az Enter művelet egyszerűbb változata. Ez a szektorhoz ad egy repülőgépet, bármiféle megszorítás-ellenőrzés nélkül.
3. In-space Egy repülőgép hívójelzését megadva, ez a Boolean-művelet igazat ad vissza, ha a repülőgép az ellenőrzött szektorban van, egyébként hamisat.
4. Occupied Megadva a magasságot, ez a Boolean-művelet igazat ad vissza, ha van repülőgép a magasság 300 méteres közelségében, egyébként hamisat.

Ezen egyszerűbb műveletek definiálásának előnye az, hogy építőkockákként használhatjuk őket, hogy összetettebb műveleteket definiáljunk a Szektor tartományhoz. A 10.8. ábrán e tartomány algebrai specifikációja látható.

Az alap konstruktorműveletek a Create és a Put, és ezeket használom a többi művelet specifikációjában. Az Occupied és az In-space ellenőrző műveletek, amelyeket a Create és a Put segítségével definiáltam, majd más specifikációkban

SZEKTOR

sort Szektor
imports INTEGER, BOOLEAN

Enter – egy repülőt ad a szektorhoz a biztonsági feltételek teljesülése esetén
Leave – a szektorból eltávolít egy repülőt
Move – egy repülőt egy magasságról egy másikra mozgat, ha az biztonságosan megtehető
Lookup – a szektorban megkeresi egy repülő magasságát

Create – egy üres szektort hoz létre
Put – egy szektorhoz ad egy repülőt a megszorítások ellenőrzése nélkül
In-space – ellenőrzi, hogy egy repülő már a szektorban van-e
Occupied – ellenőrzi, hogy egy adott magasság szabad-e

Enter (Szektor, Hívójel, Magasság) → Szektor
Leave (Szektor, Hívójel) → Szektor
Move (Szektor, Hívójel, Magasság) → Szektor
Lookup (Szektor, Hívójel) → Magasság

Create → Szektor
Put (Szektor, Hívójel, Magasság) → Szektor
In-space (Szektor, Hívójel) → Boolean
Occupied (Szektor, Magasság) → Boolean

Enter (S, HJ, M) =
    if In-space (S, HJ) **then** S **exception** (A repülő már a szektorban van)
    elseif Occupied (S, M) **then** S **exception** (Magasságütközés)
    else Put (S, HJ, M)

Leave (Create, HJ) = Create **exception** (A repülő nincs a szektorban)
Leave (Put (S, HJ1, M1), HJ) =
    if HJ = HJ1 **then** S **else** Put (Leave (S, HJ), HJ1, M1)

Move (S, HJ, M) =
    if S = Create **then** Create **exception** (Nincs repülőgép a szektorban)
    elsif not In-space (S, HJ) **then** S **exception** (A repülő nincs a szektorban)
    elsif Occupied (S, M) **then** S **exception** (Magasságütközés)
    else Put (Leave (S, HJ), HJ, M)

-- NO-HEIGHT egy konstans, ami azt jelzi, hogy nem adható vissza érvényes magasság

Lookup (Create, HJ) = NO-HEIGHT **exception** (A repülő nincs a szektorban)
Lookup (Put (S, HJ1, M1), HJ) =
    if HJ = HJ1 **then** M1 **else** Lookup (S, HJ)

Occupied (Create, M) = false
Occupied (Put (S, HJ1, M1), M) =
    if (M1 > M **and** M1 - M <= 300) **or** (M > M1 **and** M - M1<= 300) **then** true
    else Occupied (S, M)

In-space (Create, HJ) = false
In-space (Put (S, HJ1, M1), HJ) =
    if HJ = HJ1 **then** true **else** In-space (S, HJ)

10.8. ábra. Egy ellenőrzött szektor specifikációja

is használom őket. Nincs elegendő hely itt arra, hogy minden műveletet részletesen kifejtsek, de kettőt tárgyalok közülük (az Occupied-ot és a Move-ot). Ezzel az ismerettel már meg kell értsük a többi művelet specifikációját is.

1. Az Occupied művelet paraméterül egy szektort és egy magasságot vár, és ellenőrzi, hogy a magasság hozzá van-e rendelve repülőgéphez. A specifikációjában ez áll:
   - Üres szektorban (amit a Create művelet hozott létre) minden magassági szint szabad. A művelet ilyenkor, a magassági paraméter értékétől függetlenül, hamis értéket ad vissza.
   - Nem üres szektornál (amelyre lett már alkalmazva a Put művelet), az Occupied művelet ellenőrzi, hogy a megadott magasság (H paraméter) 300 méteres közelségében van-e azon repülőgép magasságának, amely legutóbb a Put művelettel a szektorhoz lett adva. Ha így van, akkor az a magasság már foglalt, így az Occupied értéke igaz.
   - Ha nem foglalt, akkor a művelet rekurzívan ellenőrzi a szektort. Ezt az ellenőrzést úgy tekinthetjük, hogy a szektorhoz utolsónak hozzáadott repülőgépre hajtódik végre. Ha a magasság nem annak a repülőgépnek a magassági tartományában van, akkor a művelet a megelőzőleg a szektorba helyezett repülőgépet ellenőrzi és így tovább. Ha éppen nincs repülőgép a megadott magasságtartományban, akkor az ellenőrzés végül az üres szektoron hajtódik végre, így hamis értékkel tér vissza.
2. A Move művelet egy repülőgépet egyik magasságból a másikba mozgat egy szektoron belül. A specifikációjában ez áll:
   - Ha a Move műveletet üres légtérre (a Create eredménye) alkalmazzák, a légtér változatlan marad és kivétel váltódik ki, ami azt jelzi, hogy a megadott repülőgép nincs a légtérben.
   - Nem üres szektorban a művelet először ellenőrzi (az In-space segítségével), hogy az adott repülőgép a szektorban van-e. Ha nem, kivétel váltódik ki. Ha igen, a művelet ellenőrzi, hogy a megadott magasság szabad-e (az Occupied művelettel), kivételt kiváltva, ha abban a magasságban már van repülőgép.
   - Ha a megadott magasság szabad, a Move művelet egyenértékű azzal, hogy a megadott repülőgép elhagyja a légteret (a Leave művelet lesz használva), és az új magasságban kerül be a szektorba (Put).

10.3. VISELKEDÉSSPECIFIKÁCIÓ

Az előző alfejezetben ismertetett algebrai technikákkal leírhatók olyan interfészek, amelyeknél az objektum műveletei függetlenek annak állapotától. Vagyis egy művelet alkalmazásának eredménye nem függhet a korábbi műveletek eredményeitől. Ahol ez a feltétel nem teljesül, ott az algebrai technikák igen sok vesződséget okozhatnak. Továbbá úgy látom, hogy méretük növekedésével a rendszer viselkedésének algebrai leírásai egyre nehezebben érthetővé válnak.

A formális specifikációknak az ipari projektekben elterjedtebb alternatívája a modellalapú specifikáció. A modellalapú specifikáció olyan formális specifikációs szemlélet, amelyben a rendszer-specifikációt a rendszer állapotmodelljeként

fejezik ki. A rendszerműveleteket specifikálhatjuk azáltal, hogy meghatározzuk, hogyan érintik azok a rendszermodell állapotát. Ezek a specifikációk együtt a rendszer egész viselkedését definiálják.

A modellalapú specifikációk fejlesztésére szolgáló nyelvek a VDM (Jones, 1980; 1986), a B (Wordsworth, 1996) és a Z (Hayes, 1987; Spivey, 1992). Itt a Z-t fogom használni. A Z-ben a rendszerek halmazokkal és a halmazok közötti relációkkal vannak modellezve. A Z ezeket a matematikai koncepciókat olyan konstrukciókkal egészíti ki, amik kifejezetten a szoftverspecifikációt támogatják.

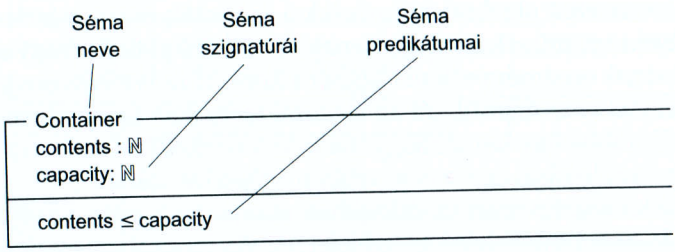
Egy modellalapú specifikációról szóló bevezetésben csupán áttekintést adhatunk a specifikáció kifejlesztésének mikéntjéről. A Z teljes leírása hosszabbra nyúlna, mint ez a fejezet. Ehelyett inkább néhány rövid példát mutatok be a technika szemléltetéséhez, és jelölést csak akkor vezetek be, amikor szükséges. A Z rendszer teljes leírása megtalálható a következő kézikönyvekben: Diller (Potter *és mások*, 1996) és Jacky (Jacky, 1997).

A formális specifikációk olvasása nehéz és unalmas lehet, különösen akkor, ha hosszú matematikai képletekkel zsúfoltak. A Z tervezői különös figyelmet fordítottak erre a problémára. A specifikációk formális leírásokkal kiegészített informális szöveg formájában adottak. A formális leírásokat rövid, könnyen olvasható szövegrészek (úgynevezett *sémák*) alakjában tartalmazzák, melyeket a csatolt szövegtől grafikus kiemelés különít el. A sémákat állapotváltozók bevezetésére használják, valamint megszorítások és műveletek megadására az állapotokon. Maguk a sémák olyan műveletekkel manipulálhatók, mint a sémakompozíció, a sémaátnevezés és a sémaelrejtés.

Hogy a formális specifikáció még hatékonyabb legyen, azt informális leírással kell kiegészíteni. A Z sémaprezentációt úgy tervezték, hogy az élesen előtűnjön a környező szövegből (10.9. ábra).

A séma szignatúrák meghatározzák a rendszer állapotát képező egyedeket, és a sémapredikátumok kijelölik azokat a feltételeket, amelyeknek ezekre az egyedekre mindig igaznak kell lenniük. Ahol a séma egy műveletet definiál, ott predikátumokkal elő- és utófeltételek jelölhetők ki. Ezek a művelet előtti és utáni állapotot definiálják. Az ezen elő- és utófeltételek közötti különbség meghatározza a műveleti sémában specifikált tevékenységet.

A Z kritikus rendszereknél történő használatának szemléltetésére elkészítettem a 3. fejezetben bemutatott inzulinadagoló vezérlőrendszerének egy formális specifikációját.



10.9. ábra. Egy Z séma szerkezete

Emlékezzünk rá, hogy a rendszer folyamatosan méri a cukorbetegek vérének cukorszintjét, és ha szükséges, automatikusan inzulint fecskendez be. Még olyan kisméretű rendszer esetén is, mint ez a készülék, a formális specifikáció meglehetősen hosszú. Bár a rendszer alapvető működése egyszerű, számos lehetséges riasztási feltételt kell figyelembe venni. Ezért csak néhányat mutatok be a rendszert definiáló sémákból; a teljes specifikáció letölthető a könyv weboldaláról.

A modellalapú specifikáció kifejlesztéséhez definiálnunk kell a specifikált rendszer állapotait modellező állapotváltozókat és predikátumokat, valamint invariánsokat (örökké igaz feltételeket) kell meghatároznunk ezekre az állapotváltozókra.

A 10.10. ábrán az inzulinadagoló állapotát modellező Z séma látható. Láthatjuk, hogyan használják a séma két fő részét. A felső részben a nevek és típusok vannak deklarálva, a séma alsó részében pedig az invariánsok.

A sémában deklarált nevek a rendszer bemeneteit, kimeneteit és belső állapotváltozóit jelölik:

1. *A rendszer bemenetei.* A Z konvenciója szerint minden input változó nevét ? követi. Egy-egy nevet deklaráltam a készülék be- és kikapcsoló gombjához (switch?), a kézi inzulinadagolás gombjához (ManualDeliveryButton?), a vércukor-érzékelő leolvasásához (Reading?), a hardvertesztelő program eredményéhez (HardwareTest?), az inzulintartály és a tű jelenlétét felismerő érzékelőkhöz (InsulinReservoir?, Needle?) és a pontos időhöz (clock?).
2. *A rendszer kimenetei.* A Z konvenciója szerint minden output változó nevét ! követi. Egy-egy nevet deklaráltam a készülék vészjelzőjéhez (alarm!), a két alfanumerikus kijelzőhöz (display1! és display2!), a pontos idő kijelzőjéhez (clock!) és a befecskendezésre kerülő inzulinadaghoz (dose!).
3. *Állapotváltozók az adag kiszámításához.* Változókat deklaráltam a készülék állapotának jelölésére (status), a korábbi vércukorszint-értékek tárolására (r0, r1, r2), a tartály kapacitásához és az éppen rendelkezésre álló inzulinmennyiséghez (capacity, insulin_available), a befecskendezett inzulinadagra kiszabott korlátokhoz (max_daily_dose, max_single_dose, minimum_dose, safemin, safemax) valamint az adag kiszámításához (CompDose és cumulative_dose). Az $\mathbb{N}$ típus nemnegatív számokat jelöl.

A sémapredikátum örökké igaz invariánsokat ad meg. A predikátum sorai között implicit „és" kapcsolat van, vagyis mindig az összes predikátumnak együtt kell teljesülnie. Néhány ezek közül egyszerű korlátot ad meg a rendszerre, míg mások a rendszer alapvető működési feltételeit határozzák meg. Néhány ezek közül:

1. Az adag legfeljebb akkora lehet, amekkora az inzulintartály kapacitása. Vagyis nem adható be több inzulin, mint ami a tartályban van.
2. Az összesített adag minden éjfélkor nullázódik. Az <1. logikai kifejezés> $\Rightarrow$ <2. logikai kifejezés> mondat jelentése szabadabban fogalmazva **ha** <1. logikai ki-

fejezés> **akkor** <2. logikai kifejezés>. Ebben az esetben az <1. logikai kifejezés> a „clock? = 000000", a <2. logikai kifejezés> pedig a „cumulative_dose = 0".

3. Az összesített adag 24 óra alatt nem lépheti túl a max_daily_dose értéket. Ha ez nem teljesül, hibaüzenet íródik ki.
4. A display2! mindig a legutóbb beadott inzulin mennyiségét, a clock! pedig a pontos időt mutatja.

INSULIN_PUMP_STATE

// Bemenetek definíciója
switch?: (off, manual, auto)
ManualDeliveryButton?: ℕ
Reading?: ℕ
HardwareTest?: (OK, batterylow, pumpfail, sensorfail, deliveryfail)
InsulinReservoir?: (present, notpresent)
Needle?: (present, notpresent)
clock?: TIME

// Kimenetek definíciója
alarm! = (on, off)
display1!: string
display2!: string
clock!: TIME
dose!: ℕ

// Az adag kiszámításához használt állapotváltozók
status: (running, warning, error)
r0, r1, r2: ℕ
capacity, insulin_available : ℕ
max_daily_dose, max_single_dose, minimum_dose: ℕ
safemin, safemax: ℕ
CompDose, cumulative_dose: ℕ

---

r2 = Reading?
dose! # insulin_available
insulin_available # capacity

// A beadott inzulinadagok összesített mennyiségét 24 óránként egyszer nullázzuk
clock? = 000000 ⇒ cumulative_dose = 0

// Ha az összesített inzulinadag túllép a megengedetten, akkor a művelet felfüggesztődik
cumulative_dose ≥ max_daily_dose ∧ status = error ⇒
display1! = "Daily dose exceeded"

// A készülék állítható paraméterei
capacity = 100 ∧ safemin = 6 ∧ safemax = 14
max_daily_dose = 25 ∧ max_single_dose = 4 ∧ minimum_dose = 1

display2! = nat_to_string (dose!)
clock! = clock?

10.10. ábra. Az inzulinadagoló állapotsémája

```
RUN
  ΔINSULIN_PUMP_STATE

  switch? = auto _
  status = running ∨ status = warning
  insulin_available ≥ max_single_dose
  cumulative_dose < max_daily_dose
  // A vércukorszint függvényében kiszámított inzulinadag
  (SUGAR_LOW ∨ SUGAR_OK ∨ SUGAR_HIGH)
  // 1. Ha a kiszámított adag 0, ne adjunk be semennyi inzulint
  CompDose = 0 ⇒  dose! = 0
  ∨
  // 2. Ha a kiszámított adagot beadva túllépnénk a maximális napi adagot, akkor a beadandó
  // mennyiség a maximális napi adag és az összesített adag különbsége lesz
  CompDose + cumulative_dose > max_daily_dose ⇒ alarm! = on ∧ status' = warning ∧ dose! =
  max_daily_dose–cumulative_dose
  ∨
  // 3. Normális helyzet. A kiszámított adag lesz beadva, ha az nem lépi túl a maximális egyszeri
  // adagot. Ha a kiszámított adag túl nagy, akkor a maximális egyszeri adag lesz beadva
  CompDose + cumulative_dose < max_daily_dose ⇒
        ( CompDose ≤ max_single_dose ⇒ dose! = CompDose
        ∨
        CompDose > max_single_dose ⇒  dose! = max_single_dose )
  insulin_available' = insulin_available–dose!
  cumulative_dose' = cumulative_dose + dose!

  insulin_available ≤ max_single_dose * 4 ⇒  status' = warning ∧
  display1! = "Insulin low"

  r1' = r2
  r0' = r1
```

10.11. ábra. A RUN séma

Az inzulinadagoló úgy működik, hogy tízpercenként ellenőrzi a vércukorszintet, és (sejthető módon) inzulint ad be, ha a vércukorszint változásának üteme gyorsul. A 10.11. ábrán bemutatott RUN séma a készülék normális működési körülményeit modellezi.

Ha a deklarációs részben sémanév szerepel, az egyenértékű azzal, mintha a sémában deklarált összes név a deklarációs részben, a feltételek pedig a predikátum részben szerepelnének. A delta séma (Δ) a 10.11. ábra első sorában ezt szemlélteti. A delta azt jelenti, hogy az INSULIN_PUMP_STATE sémában definiált állapotváltozók látszanak a RUN séma hatáskörében, akárcsak azok a változók, amik a műveletek előtti és utáni állapotértékeket hordozzák. Utóbbiakat úgy jelöljük, hogy az INSULIN_PUMP_STATE sémában definiált név után felsővesszőt írunk. Az insulin_available egy művelet előtt, az insulin_available' pedig az után elérhető inzulinmennyiséget jelöli.

A RUN séma a rendszer működését definiálja, olyan predikátumokkal, melyek rendes működés esetén adnak igaz értéket. Természetesen ezek kiegészítik az INSULIN_PUMP_STATE sémában definiált invariáns (mindig igaz) predikátu-

mokat. Ez a séma szintén bemutatja a Z egyik lehetőségének – a sémakompozíciónak – a használatát, amellyel a SUGAR_LOW, SUGAR_OK és SUGAR_HIGH sémákat nevük megadásával beépítjük a sémába. Vegyük észre, hogy a három séma „vagyolva van", vagyis mindhárom lehetséges körülményhez van egy külön séma. A sémák kompozíciója azt jelenti, hogy a specifikációt apróbb részekre robbanthatjuk szét ugyanúgy, ahogy függvényeket vagy metódusokat definiálunk egy programban.

Nem megyek bele a RUN séma részleteibe, de a lényeg, hogy az elején olyan predikátumokat definiál, amelyek a normális működés esetén igazak. Például kimondja, hogy a rendes működésnél a maximális egyszeri adagnál többnek kell lennie a készülékben. A három, különböző vércukorszintet ábrázoló sémát azután vagyolja, és ahogy később látszik, ezek definiálják a CompDose állapotváltozó értékét.

A CompDose értéke a vércukorszint alapján kiszámított, beadandó inzulin mennyiségét jelöli. A séma hátralevő predikátumai különféle ellenőrzéseket határoznak meg, amik garantálják, hogy a ténylegesen beadandó adag (dose!) megfelel a rendszer biztonsági előírásainak. Például az egyik biztonsági előírás az, hogy egyetlen inzulinadag sem léphet túl egy bizonyos előre megadott maximális értéket.

Végül az utolsó két predikátum az insulin_available és a cumulative_dose értékének változását definiálja. Figyeljük meg, hogyan használtam a nevek vesszős változatait!

Az utolsó sémás példa a 10.12. ábrán szerepel. Ez az inzulinadag kiszámítását határozza meg, azt feltételezve, hogy a beteg vérének cukorszintje valamilyen biztonságos korlátok között van. Ilyen körülmények között csak akkor lesz inzulin beadva, ha a vércukorszint emelkedik, és a szint változásának üteme is gyorsul. A SUGAR_LOW és SUGAR_HIGH sémák arra az esetre határozzák meg az adagot, ha a cukorszint ezen a biztonsági zónán kívül esik. A séma predikátumai a következők:

SUGAR_OK

```
r2 ≥ safemin ∨ r2 ∧ safemax
// stabil vagy csökkenő cukorszint
r2 ≤ r1 ⇒ CompDose = 0
∨
// a cukorszint növekszik, de a növekedés üteme lassul
// ha a kiszámított adag kerekítve nulla, a minimális adagot kell beadni
r2 > r1 ∧ (r2-r1) < (r1-r0) ⇒ CompDose = 0
∨
// a cukorszint növekszik és a növekedés üteme is nő,
r2 > r1 ∧ (r2-r1) ≥ (r1-r0) ∧ (round ((r2-r1)/4) = 0) ⇒
                CompDose = minimum_dose
∨
r2 > r1 ∧ (r2-r1) ≥ (r1-r0) ∧ (round ((r2-r1)/4) > 0) ⇒
                CompDose = round ((r2-r1)/4)
```

10.12. ábra. A SUGAR_OK séma

1. Az első predikátum a biztonsági zónát határozza meg; vagyis r2-nek a safemin és safemax értékek közé kell esnie.
2. Ha a cukorszint stabil vagy süllyed [amit az jelez, hogy r2 (a későbbi leolvasás) kisebb vagy egyenlő mint r1 (az előző leolvasás)], akkor a beadandó inzulinmennyiség nulla.
3. Ha a cukorszint emelkedik (r2 > r1), viszont az emelkedés üteme lassul, a beadandó adag nulla.
4. Ha a cukorszint emelkedik és az emelkedés üteme állandó, akkor a minimális adag lesz beadva.
5. Ha a cukorszint emelkedik, és az emelkedés üteme gyorsul, akkor a beadandó inzulin mennyisége egyszerű képlettel lesz meghatározva a mért értékek alapján.

A rendszer időbeli viselkedését (vagyis azt a tényt, hogy a vércukorszint-érzékelő minden tíz percben megvizsgálásra kerül) nem modellezem a Z segítségével. Bár ez bizonyosan lehetséges, ugyanakkor nehézkes is, és úgy vélem, hogy informális leírással a specifikáció sokkal tömörebben megadható, mint formális specifikációval.

Kulcsfogalmak

- A formális specifikációs módszerek az informális követelményspecifikációs technikák kiegészítői. A specifikációbeli félreérthetőségek tisztázása végett természetes nyelvű követelménydefinícióval együtt is használhatók.
- A formális specifikációk pontosak és egyértelműek. Eloszlatják a specifikációk kétségeit, és elkerülnek bizonyos nyelvi félreértések okozta problémákat. Ugyanakkor a nem specialisták számára a formális specifikációk megértése nehéz lehet.
- A formális módszerek fő értéke a szoftverfolyamatban az, hogy a rendszerkövetelmények elemzését már egy korai szakaszban kikényszerítik. A hibák kijavítása ekkor olcsóbb, mint egy leszállított rendszer módosítása.
- A formális specifikációs technikák a kritikus rendszerek fejlesztésénél a leginkább költséghatékonyak, ahol a biztonságosság, megbízhatóság és a védelem különösen fontosak. Szabványok megadásához is használhatóak.
- A formális specifikációk algebrai technikái különösen alkalmasak interfészek megadására, ahol az interfészek objektumosztályok vagy absztrakt adattípusok halmazaként vannak definiálva. Ezek a technikák elrejtik a rendszer állapotát, és a rendszert az interfészműveletek közötti kapcsolatok szempontjából definiálják.
- A modellalapú technikák a rendszert olyan matematikai konstrukciók segítségével modellezik, mint a halmazok és a függvények. Kiemelhetik a rendszer állapotát, ami egyszerűsít bizonyos típusú viselkedési specifikációkat.
- Modellalapú specifikációban a műveleteket a rendszer állapotára tett elő- és utófeltételekkel definiáljuk.

További irodalom

„Correctness by construction: Developing a commercially secure system". Jó leírás arról, hogyan használhatók a formális módszerek biztonságkritikus rendszerek fejlesztésénél. (Hall, A. és Chapman, R., *IEEE Software*, 19 (1), January 2002.)

IEEE Transactions on Software Engineering, January 1998. A folyóiratnak ez a száma külön szekciót tartalmaz a formális módszerek gyakorlati használatáról a szoftvertervezésben. A Z-ről és a LARCH-ról is szerepelnek írások benne.

„Formal methods: Promises and problems". A cikk valósághűen tárgyalja a formális módszerek használatával nyerhető előnyöket és a formális módszerek használatának a gyakorlati szoftverfejlesztésbe való integrálásának nehézségeit. (Luqi és Goguen, J., *IEEE Software*, 14 (1), January 1997.)

Feladatok

10.1. Mondjon okokat, miért érdemes megelőznie a rendszer tervezését formális specifikáció kifejlesztésének.

10.2. Azt a feladatot adták önnek, hogy „adjon el" formális specifikációs technikákat egy szoftverfejlesztéssel foglalkozó cégnek. Körvonalazza, hogyan magyarázná el a formális specifikációk előnyeit szkeptikus, szakmában jártas szoftvermérnököknek.

10.3. Magyarázza meg, miért különösen fontos az alrendszerinterfészeket precíz módon definiálni, és miért különösen alkalmas az algebrai specifikáció az alrendszerinterfészek specifikációjára.

10.4. Egy vermet reprezentáló absztrakt adattípust a következő műveletekkel láttak el:

New:	Új vermet hoz létre.
Push:	Egy elemet a verem tetejére tesz.
Top:	Lekérdezi a verem tetején lévő elemet.
Retract:	Eltávolítja a verem tetején lévő elemet, és a módosított veremmel tér vissza.
Empty:	Igaz, ha már nincs elem a veremben.

Definiálja ezt az absztrakt adattípust algebrai specifikáció segítségével.

10.5. Egy ellenőrzött légtérszektor példájánál az a biztonsági feltétel, hogy a repülőgépek nem lehetnek egymás magasságának 300 méteres közelségében ugyanazon szektoron belül. Módosítsa a 10.8. ábrán látható specifikációt úgy, hogy az megengedje, hogy több repülőgép ugyanabban a magasságban legyen egy szektoron belül, ha azok horizontális távolsága legalább 8 km. A szomszédos szektorokban lévő repülőgépeket figyelmen kívül hagyhatja. *Javaslat:* Módosítania kell a konstruktorműveleteket, hogy azok a repülőgép pozícióját és magasságát is tartalmazzák. Szintén meg kell adnia egy olyan műveletet, amely két pozíciót megkapva visszaadja a köztük lévő távolságot.

10.6. A bankautomaták az ügyfél kártyáján lévő információra támaszkodnak, melyek megadják a banki azonosítót, a számlaszámot és az ügyfél személyi azonosító kódját. Szintén hozzájutnak a számlainformációkhoz egy központi adatbázisból, és a tranzakció végeztével frissítik az adatbázist. Az ATM működése ismeretének felhasználásával írjon olyan Z sémákat, amelyek definiálják a rendszer állapotát, a kártya érvényességének ellenőrzését (ekkor az ügyfél azonosítója is ellenőrzésre kerül) és a készpénzfelvételt.

10.7. Módosítsa a 10.10. ábrán látható inzulinadagolós sémát egy újabb biztonsági feltétel megadásával, hogy a ManualDeliveryButton? csak akkor kaphasson nemnulla értéket, ha a készülék kapcsolója a kézi pozícióra van állítva.

10.8. Írjon egy SELF_TEST nevű Z sémát, amely teszteli az inzulinadagoló hardverelemeit, és beállítja a HardwareTest? változó állapotát. Ezután módosítsa a RUN sémát úgy, hogy az az inzulin beadása előtt ellenőrizze a hardver megfelelő működését. Ha a teszt sikertelen, akkor az adag legyen nulla, és a készülék jelezze kijelzőjén a hibát!

10.9. A Z ismeri a sorozat fogalmát, amely hasonló a tömbhöz. Például egy S sorozat esetén annak elemeit S[1], S[2] stb. módon érhetjük el. Egy sorozat elemeinek számát a # operátorral határozhatjuk meg. Tehát, ha az S sorozat értéke [a, b, c, d], akkor #S értéke 4. Az S sorozat elejéhez az (a + S), a végéhez az (S + a) kifejezéssel fűzhetünk új elemet. Írjuk meg ezekkel a konstrukciókkal a 10.7. ábrán algebrailag leírt láncolt lista specifikációját Z-ben!

10.10. Ön rendszermérnök, és arra kérik, hogy mondja meg, mi a legjobb módja egy szívritmusszabályzó biztonságosságkritikus rendszere kifejlesztésének. Ön a rendszer formális specifikációját javasolja, de javaslatát a főnöke visszautasítja. Úgy látja, hogy főnöke érvei gyengék és előítéleteken alapulnak. Etikus-e a rendszert olyan módszerek segítségével fejleszteni, amelyekről azt gondolja, hogy nem alkalmasak arra?