
# Rendszerintegráció és -felügyelet laboratórium (VIMIM309)

## Szabályalapú üzleti logika

Mérési segédlet

Készítette: Bergmann Gábor

Utolsó módosítás: 2011. április 7.

Verzió: 1.0

Budapesti Műszaki és Gazdaságtudományi Egyetem
Méréstechnika és Információs Rendszerek Tanszék

# 1 Bevezető

A labor során a méréseket végző hallgató a gyakorlatban is megismerkedik a rendszerintegráció és rendszerfelügyelet során használatos módszerekkel és eszközökkel. Végigköveti egy elosztott alkalmazás megvalósításának és felügyeletének legfontosabb lépéseit, ipari környezetben használt integrációs köztes réteg (middleware) technológiák és felügyeleti eszközök használatával. A mérések a következő témakörökhöz kapcsolódnak:

1. Munkafolyamatok megvalósítása Java nyelven
2. Megbízható üzenetküldés IBM WebSphere MQ alapon
3. Kommunikáció JMS és JMX technológia segítségével
4. OSGi szolgáltatások fejlesztése
5. Modell alapú eszközintegráció elosztott környezetben (SDE)
6. Szabályalapú üzleti logika
7. Üzleti folyamatok felügyelete

A megelőző méréseken a hallgatók különféle elosztott szolgáltatásintegráció megoldásokkal valósították meg egy mintarendszer üzleti logikáját. A jelen mérés során az elosztottság helyett a deklarativitás lesz fókuszban: szabályalapú formalizmussal kell a korábbi mérésekben vizsgált üzleti logikát megvalósítani, a JBoss Drools Expert terméke segítségével. Imperatív programkód helyett egy ha–akkor jellegű szabályokból álló szabálytárnak kell a korábbiakhoz hasonló működést tanúsítania. A hallgatók elsajátíthatják a szabály alapú produkciós rendszerek alapelveit, és megismerkedhetnek a szabály alapú üzleti logika rendszerek használatával.

# 2 Háttérismeretek

## 2.1 Produkciós rendszerek alapfogalmai

Az informatika legkülönbözőbb területein elterjedtek a különféle *szabály alapú* (rule based) megoldások, a tűzfalak konfigurációjától a MAKEFILE-okig. Közös jellemzőjük, hogy imperatív programkód helyett deklaratívan megjelölt szabályok értelmezése alapján történik a vezérlés.

Először a szabály alapú szakértői rendszerek (rule based expert systems) megvalósítására születtek meg a szabály alapú következtető rendszerek (inference rule systems). Ezek a szakértői rendszerek a világról alkotott ismereteket egy *tudásbázisban* (knowledge base) tárolták, amely kész tényállításokat tartalmazó *ténybázisból* (fact base) és következtetési szabályokat tartalmazó *szabálybázisból* (rule base) áll. A következtetési szabályok *ha–akkor* alakúak, ahol a *feltétel rész* vagy *bal oldal* (precondition, left hand side, LHS) a ténybázis elemeire épített logikai feltétel (jellemzően szabad változókkal), a *következmény rész* vagy *jobb oldal* (postcondition, right hand side, RHS) pedig egy ebből kikövetkeztethető új tény; például ilyen szabály az Influenzás → Köhög.

Fontos alapgondolat a szabályok láncolódása, vagyis hogy (esetleg további tényekkel együtt) maga a következmény is teljesítheti egy szabály feltételrészét, amelynek újabb következményei lesznek, stb. A hátrafelé következtető (backward chaining) rendszerek a következmények alapján próbálják több lépésen keresztül meghatározni az ismeretlen előfeltételeket, például a diagnózis alapján a betegség eredeti okát; ezzel analóg elven működik pl. a Prolog. Az előre következtető (forward chaining) rendszerek az eleve adott tényekből kiindulva újabb és újabb tényeket bizonyítanak be a szabályok alapján; például egy fertőzés lehetséges tüneti következményeinek előjelzésére.

A továbbiakban kizárólag előre következtető rendszerekkel foglalkozunk. Mivel a ténybázis a szabályok alkalmazása miatt folyamatosan változik, *munkamemóriának* (working memory, WM) is nevezik. A szabályok képesek új tényeket beszúrni a WM-be; gyakran *produkciós szabályoknak* (production rules) hívják őket, a teljes rendszert pedig produkciós rendszernek (production rule system). Ha egy produkciós szabály minden feltétele ki van elégítve, akkor a szabály aktivált (activated, triggered). Mivel

a feltételrészben jellemzően szabad (lekötetlen) változók vannak, a feltételeket valójában ezen változók egy konkrét lekötése (behelyettesítése) elégíti ki; ezeket a kielégítő behelyettesítéseket nevezzük *aktivációnak* (activation), és a változók konkrét értékeiből formált tuple („n-nes") azonosítja őket. A változók egy konkrét behelyettesítése alapján értelmezett RHS a szabály *példányosítása* (instantiation), és a szabálypéldány tényleges végrehajtása a *tüzelés* (firing). Az összes érvényes aktiváció halmaza a napirend (*agenda* vagy conflict set), ezek közül kell kiválasztani a tüzelendőt.

Produkciós rendszerekben általános alapelv, hogy az előre következtetés addig tart, amíg nincs már több alkalmazható szabály. Egy szabálypéldány akkor alkalmazható, ha még eltüzeletlen, de érvényes az aktivációja. A következtetés lezárultával az alaptények megváltoztathatóak, ez alapján egy új következtetési fázisban újabb szabálypéldányok tüzelhetőek. Egyúttal korábban eltüzelt de érvénytelenné vált szabálypéldányok visszavonhatódnak, az általuk beszúrt (assert / insert) tények kikerülnek (retract) a WM-ből.

A mesterséges intelligencia egy területe a szabály alapú szakértői rendszerek kutatása. Az évek során megszülettek a klasszikus nyelvek és implementációk (pl. OPS5) és *inkrementális mintaillesztő* algoritmusok (pl. RETE, TREAT, LEAPS). Utóbbi algoritmusok célja, hogy nagy tudásbázisokat is kezelhetővé tegyenek azáltal, hogy nem értékelik ki minden tüzelés után újra és újra az összes szabály feltételrészét a teljes tudásbázison. Ehelyett a tüzelés hatására történt módosítások alapján az új aktivációkat és az érvénytelenné válókat keresik meg, inkrementálisan karbantartva az agendát.

## 2.2 Szabályalapú üzleti logika

Vannak olyan alkalmazási területek, ahol a megszokott technikáktól eltérően célszerű felépíteni az üzleti logikát. Egy érdekes lehetőség „kiszervezni" a hagyományos imperatív stílusban készült programrészekből egy feljebb vázolt szabály alapú, deklaratív formalizmusba. A koncepció szerint az üzleti logikát *üzleti szabályok* (business rule) definiálják, amelyek az *üzleti objektumokat* (business object) figyelhetik és manipulálhatják, és egy *üzleti szabálymotor* (business rule engine, BRE) hajtja őket végre. A hagyományosan elkészített programrészek az üzleti objektumok írásán és olvasásán keresztül érintkezhetnek a szabálymotorral, illetve a szabályok akció része is hívhat imperatív kódot.

Az üzleti szabályok ha-akkor jellegű feltételek alapján végeznek akciókat az üzleti objektumok összességén. A tipikus szabály egy objektumon vizsgál meg bizonyos feltételeket, majd ez alapján módosíthat attribútumokat. Az irodalomban olvasott klasszikus példák általában árajánlati vagy díjszabási logikát írnak le: „ha az ügyfél 30 év alatti, emeljük 35%-al az ajánlatot" vagy „ha az ügyfél egyenlege 500Ft alá csökkent, értesítsük". Ismerősebb példa lehet egy hallgatói adminisztrációs rendszer: „ha a hallgatónak legalább húsz lezárt féléve van, nem szerzett aláírást diplomatervezésből és nem kapott köztársasági elnöki engedélyt, akkor megszüntetendő a jogviszonya, feltéve hogy ötéves képzésre jár és az ezt előíró jogszabály hatályba lépése óta kezdte tanulmányait". Léteznek emellett bonyolultabb esetek is, ahol egynél több üzleti objektumra vonatkozik egy feltétel, pl.: „ha volt már korábban másik ügyfél, akinek azonos a lakcíme, akkor nem adható kedvezmény". Azonos szerkezetű (általában egyszerű), csak paraméterekben különböző szabályok nagy mennyiségben definiálhatóak szabálysablonok segítségével, vagy akár egy döntési tábla (pl. megfelelően formázott Excel spreadsheet fájl) formájában.

A BRE tulajdonképpen egy produkciós rendszer a logikai következtetés matematikai hátterétől elvonatkoztatva, gyakorlatilag programozási platformként. A szabályok RHS részét *akció*nak is hívják, mivel nem csak új tények (objektumok) kikövetkeztetését és WM-be beszúrását jelentheti, hanem akár tények törlését/visszavonását is, sőt, gyakorlatilag tetszőleges programkódot tartalmazhat. Ugyan az akció lehet imperatív szerkezetű, ennek ellenére a teljes program vezérlési struktúráját továbbra is a deklaratívan definiált szabályok adják meg. A szabályok jobb oldalán az új tények beszúrása szintén akció jellegű; tehát ha az eredeti szabályaktiváció megszűnik, a beszúrt objektum ettől még a WM-ben marad. Viszont általában lehetőség van az RHS részben a logikai következtetők működéséhez hasonló *logikai beszúrást* (logical assert) kérni; a végrehajtott logical assert automatikusan visszavonódik az aktiváció megszűntekor. Mivel nem logikai következtetés jellegűek az akciók, a szabály minden érvényes aktivációra alkalmazható, nem csak a még eltüzeletlenekre. Ez azt jelenti, hogy egy szabálypéldány nem csak egyszer tüzelhet; ezért a szabálynak meg kell szüntetnie saját aktivációját, nehogy végtelen ciklus alakuljon ki; bizonyos rendszerekben ez automatikusan megtörténik (vagyis az

agenda listáról eltűnik az aktiváció, hiába maradnak kielégítve a feltételek). Ha több aktiváció is tüzelhető egyszerre, akkor a választás nemdeterminisztikus, de szabályprioritásokkal befolyásolható.

A BRE alapú architektúra alkalmazásának egyik motiválója lehet, ha arra van szükség, hogy üzleti döntéshozók is áttekinthessék az automatizált üzleti döntés működési elveit. Jó esetben ez teljesül, és ideális esetben akár módosíthatják is a szabályokat. Ehhez a célkitűzéshez komoly támogatást nyújtanak a BRE rendszerek (természetes nyelvi verbalizáció, spreadsheet alapú döntési tábla, stb.).

Fontosak azok az alkalmazási területek is, ahol a szabályok nagyon gyakran változnak (a többi programrésznél sokkal változékonyabbak), például bármikor bevezethető egy új ajánlattételi stratégia, vagy könnyen kezelhető a hatósági rendeletek bonyolult kivételrendszereinek gyors változása. Nem kell minden változtatáson szoftverfejlesztő szakembereknek dolgozniuk, majd a szoftvertelepítést frissíteni, stb.; az üzleti logika a szabálybázisban könnyen módosítható. Erre szintén fejlett eszköztár áll a rendelkezésre: szabálytár szerverek, verziókezelési támogatás, integrált fejlesztőkörnyezet, stb. Ezen eszközök összessége a Business Rule Management System (BRMS).

Járulékos kedvező hatásnak mondható, hogy a programfejlesztők az üzleti logika jobb elkülönítésére kényszerülnek (pl. nem szóródik szét az üzleti logika a megjelenítési rétegtől az adatelérési vagy kommunikációs modulokig, esetleg redundánsan), aminek számos architekturális előnye lehet.

A megközelítés hátránya viszont, hogy a szabály alapú nyelveken nehézkesebb megfogalmazni olyan munkafolyamatokat, ahol a sorrendiségnek nagy szerepe van. Ezen kívül itt is megjelenhetnek a klasszikus absztrakciós problémák: a szabálynyelv vagy túl absztrakt, és nem képes minden szükséges logikát kifejezni, vagy ellenkező esetben annak a veszélye fenyeget, hogy nem is egyszerűbb az imperatív programnál; mindemellett felléphet absztrakciós szivárgás (leaky abstraction). a felsorolt okokból nem mindenhol pozitívak a szabály alapú üzleti logika rendszerekkel szerzett hosszú távú tapasztalatok.

A népszerű Business Rule termékek közé tartozik a JBoss Drools, az (immár IBM) ILOG JRules, a Microsoft BizTalk Business Rules Framework és a FICO Blaze Advisor. Különleges igények kielégítésére léteznek specializált szabály alapú rendszerek, mint pl. a valósidejű vezérlési feladatra szánt Gensym G2. A különféle gyártók többé-kevésbé hasonló szolgáltatásokat kínálnak; a továbbiakban a nyílt forrású Drools rendszerrel foglalkozunk.

## 2.3 JBoss Drools

A JBoss cég nyílt forrású Drools termékcsaládja Java alapú (JavaBean) üzleti objektumokat manipuláló üzleti szabályok fejlesztésére, végrehajtására, üzemeltetésére nyújt támogatást. A jelenlegi Drools 5 platform a következő fontosabb részprojektekből épül fel:

- **Drools Expert**. Ez a központi szerepű modul a tényleges üzleti szabály végrehajtó rendszer. Tartalmazza a szabályleíró nyelv(ek)et, a végrehajtó szabálymotort a megfelelő programozói interfészekkel, továbbá az Eclipse alapú fejlesztőkörnyezetet.
- **Drools Guvnor**. A Drools BRMS rendszere. Szerveroldali modulja (Java webalkalmazás) szabályok és egyéb erőforrások megfelelően rendszerezhető központi tárolását végzi verziókezelési és hozzáférésvédelmi támogatással. Webes és Eclipse-be épülő felhasználói felülete mellett különféle szabványos protokollokon keresztül is elérhető a tartalma, illetve a tárolt szabályok természetesen futtathatóak Drools szabálymotorral.
- **Drools Flow**. Mivel a szabályalapú formalizmus a sorrendiség és vezérlési folyam kifejezésében gyenge, az *üzleti folyamat* (workflow) formalizmusok pedig épp ezt biztosítják, kézenfekvő kiterjesztés egy munkafolyamat végrehajtó motor implementálása a Drools BRE rendszer fölé. Leíró fájlokban definiált folyamatstruktúra (szekvencia, elágazás, párhuzamosság) csomópontjaiban az üzleti szabályok egy-egy csoportja (ruleflow group) hajtható végre a szokásos módon. Természetesen imperatív vagy manuális végrehajtású csomópontok is rendelkezésre állnak.
- **Drools Fusion**. Komplex esemény-feldolgozást (Complex Event Processing) végző motor. Különféle forrásokból különböző időpontokban esemény objektumok érkezhetnek, melyekre

temporális feltételekkel kiegészített Drools szabályok reagálhatnak. A régi események automatikusan megőrződnek mindaddig, amíg a szabályok szempontjából relevánsak lehetnek.

- **Drools Planner**. Tervezési és optimalizációs problémák szabály alapú definiálására és intelligens megoldására szolgáló termék. A Drools szabályokkal kifeszíthető keresési térben a mesterséges intelligencia területéről ismert módszerekkel keres (optimális) megoldást.

Ezek közül a jelenlegi mérés során egyedül az alapvető Drools Expert modulra lesz szükségünk.

## 2.4 Drools Expert alapok

A Drools Expert használatához erősen ajánlott a (sajnos nem teljesen friss) felhasználói dokumentáció ismerete és használata, továbbá a Hello World tartalmú Drools projekt létrehozása és tanulmányozása.

A Drools terminológiában a tudásbázisnak (Knowledge Base) nem része a ténybázis, csupán a szabálybázis (és egyéb elemek, pl. rule flow leírás). Tényleges szabályvégrehajtáshoz ezért kérni kell egy úgynevezett sessiont a tudásbázisból, amely már tartalmazni fog ténybázist, így beszúrhatóak üzleti objektumok és tüzelhetnek szabályok. Egy tudásbázis fölé több független session is nyitható egyszerre.

Az állapotmentes (stateless) session egyszerű működésű: nem rendelkezik „emlékező” WM-mel, hanem egyszeri végrehajtáskor átadható egy vagy több objektum, amely a ténybázist fogja alkotni. Ezek után minden rájuk alkalmazható szabály tüzelni fog, és a működés láncolás nélkül befejeződik. Assert vagy retract művelet nem használható; az objektumokon végzett módosítások nem változtatják az aktivációs listát. Ilyen session csak nagyon egyszerű esetben használható, amikor egy vagy néhány objektumot egymásrahatás nélküli szabályokkal szeretnék feldolgozni.

A mi céljainknak az állapotos (stateful) session fog megfelelni, ezért a továbbiakban csak ezzel foglalkozunk. Itt van valódi WM, amelybe beszúrhatóak, ill. kivehetőek tények. Ezek után meghívható a `fireAllRules()` metódus, amely láncolással hajtja végre a szabályokat, folyamatosan módosítva közben a WM tartalmát; akkor tér vissza a metódus, ha nincs már tüzelhető szabály. Ezek után a WM „kívülről” tovább módosítható, majd újra hívható a `fireAllRules()`, stb.

Mivel a Drools is inkrementális mintaillesztőt használ, az agenda frissítéséhez szükséges értesülnie a WM változásiról. A WM-be beszúrást és törlést természetesen a Drools API-jain keresztül bonyolítjuk, ezért ez nem okoz gondot. Az objektumok módosítása azonban Java környezetben nem detektálható egyszerűen, ezért ezt a Drools számára jelezni kell valamilyen módon. A megoldás az objektum beszúrásakor az `insert()` metódustól visszakapott `FactHandle` referencia, melyen keresztül értesíthető a Drools Engine a módosításról, hogy újraolvashassa az objektum attribútumait. Szerencsére a leggyakoribb esetben, a szabály akció részében történő módosítások esetén ennél elegánsabb megoldás is van (ld. később).

Egy nagyon egyszerű Drools-os programnak definiálnia kell az üzleti objektumok adatmodelljét; ez Java osztályokat jelent, amelyek a JavaBean konvenciókhoz igazodnak, pontosabban `getXXX()` és `setXXX()` metódusokon keresztül attribútumokat kínálnak fel (az objektumok mezői a Drools számára nem láthatóak). Ezen felül szükség van a szabályokat definiáló fájlra (alapesetben .drl formátumban). A keretprogram a szabálydefiníciós fájl alapján felépíttet egy Drools tudásbázist, majd kér egy állapotos sessiont a `newStatefulKnowledgeSession()` metódussal. A session WM-jébe `insert()` metódussal elhelyezi a kezdeti üzleti objektumokat, majd meghívja a `fireAllRules()` metódust.

A szabályok definiálására a Drools alapértelmezetten a DRL szöveges nyelvet használja; emellett létezik az űrlapok útján szerkeszthető BRL szabályleíró (Guided Rule). A saját szakterület-specifikus (DSL) szabályleíró nyelvek specifikálása is támogatott, amelyek segítségével üzleti döntéshozók is könnyebben megérthetik, módosíthatják a szabályokat. Hasonló célból lehetőség van döntési táblák használatára is: az Excel füzetként ábrázolt döntési tábla minden sora egy-egy hasonló szerkezetű szabályt paraméterez (jobb- és bal oldalon egyaránt). Külső konfiguráció céljaira létezik egy ChangeSet XML formátum is, amely több másik szabályleíró erőforrást (akár hasonló XML fájlt) nevezhet meg; ilyen XML szolgáltatást biztosít a Guvnor szerver is. A továbbiakban csak a DRL formátummal foglalkozunk.

## 2.5 Drools szabálynyelv (DRL)

A Java osztályokhoz hasonlóan a Drools szabályok is csomaghierarchiába sorolhatóak, ezért a DRL fájl ugyanúgy **package** deklarációval kezdődik, mint egy .java forrásfájl. A hasonlóságot folytatva ezek után **import** deklarációk következnek. Fontos, hogy ezek nem Drools szabályokat, hanem Java osztályokat importálnak a névtérbe, mivel a szabályok feltételrésze Java osztályokra hivatkozik, és a tetszőleges Java utasításokat tartalmazó akciók definiálásánál is hasznosak lehetnek. Ezek után következhetnek a szabályok definíciói.

A szabálydefiníciót a **rule** kulcsszó vezeti be, amelyet a szabály idézőjelek közé tett neve követi. Ezek után igény szerint a szabály működését befolyásoló *opciók* specifikálhatóak. A szabály két legfontosabb része a **when** kulcsszó után következő feltételrész és a **then** kulcsszó után megadott akciólista. A szabálydefiníciót **end** zárja. Példaként nézzük meg a projektvarázsló által létrehozott HelloWorld példa egyik szabályát!

```
rule "GoodBye"
      when
            Message( status == Message.GOODBYE, myMessage : message )
      then
            System.out.println( myMessage );
end
```

A GoodBye nevű szabály feltételrésze egy `Message` típusú üzleti objektum létezését írja elő, amelynek a `status` attribútumára igaz egy feltétel (jelen esetben egyszerű egyenlőség). A `message` nevű attribútumra (nem keverendő a `Message` osztállyal) nem mond ki feltételt az LHS, de értékét egy `myMessage` nevű Drools változóba tárolja. Az akció rész ezek után kiírja ezen változó értékét. Összességében tehát ez a szabály minden `Message.GOODBYE` státuszú üzenetet kiír.

Az LHS-ben objektumok vagy attribútumaik elé fűzött kettőspont operátorral definiálhatóak Drools változók (mint a fenti példában a `myMessage`). Ezek az LHS változók szerepelnek a szabály aktivációiban. Ezen felül természetesen definiálhatóak *lokális változók* az RHS-ben is; ezek természetesen nem részei az aktivációnak, hiszen csak tüzelés közben kapnak értéket. Hogy a Drools változóit ne keverjük össze a Java elemekkel, '`$`' karakterrel kezdődő nevet szokás választani nekik (ezt a Drools megengedi), bár a HelloWorld példa nem élt ezzel a lehetőséggel.

Amint a fenti példa is mutatja, az akció blokkba egyszerű Java utasítások írhatóak (metódushívás, lokális változódefiníció és értékadás, példányosítás, vezérlési szerkezetek, stb.). Ezen felül a Drools biztosít néhány különleges műveletet. Az **insert**`(obj)` és **retract**`(obj)` parancsok a paraméterül kapott objektumot beszúrják ill. kiveszik a WM-ből. Az **insertLogical**`(obj)` logikai beszúrást hajt végre, amely visszavonódik, ha a feltételrész érvényét veszti. A korábban tárgyalt okok miatt az attribútumok manipulációja (vagy ilyen hatással járó metódushívások) után **update**`(obj)` hívandó, hogy a Drools is értesülhessen a változásról. Ennek egy elegánsabb alternatívája a **modify**`(obj){...}` blokk, amelynek a végén ez automatikusan megtörténik; a blokkon belül pedig az objektum metódusai kényelmesebben hívhatóak, '`obj.`' prefix nélkül.

A feltételrészt többnyire objektumminták alkotják. Az *objektumminta* egy osztálynév, amelyet zárójelezett listában az attribútumaira vonatkozó *korlátozások* követnek. A Drools mintaillesztője megfelelő típusú objektumpéldányokat keres a WM-ben, amelyek (együttesen) kielégítik a korlátozásokat. Azért együttes kielégítésről van szó, mert kettőspont operátorral prefixálva egyrészt (a) magukat a mintában szereplő objektumokat, másrészt (b) a korlátozásokban használt attribútumokat meg lehet kötni Drools változóba, és egy másik objektumminta korlátozásaiban felhasználni. Az objektumminták mellett **eval**`(...)` ellenőrző kifejezések is megadhatóak további őrfeltételként.

A feltételrészben szereplő feltételek (objektumminta, eval) alapértelmezésben és-kapcsolatban állnak, de **and** és **or** operátorokkal ill. zárójelekkel ez befolyásolható. Nagy a jelentősége a **not** kvantornak, amely az utána következő feltétel (vagy feltételek zárójelezett összessége) *kielégíthetetlenségét* mondja ki, vagyis: nincsenek olyan elemek a WM-ben, amelyek megadott típusúak és megadott viszonyban állnak egymással és a többi objektummal. Hasonló az **exists** kvantor, amely egzisztenciális jelleggel kielégíthetőséget mond ki; ha egy feltétel elé helyezzük, akkor nem fog számítani, hányféleképpen elégíthető ki a feltétel, csak hogy kielégíthető-e. Végül a **forall** kvantor azt mondja ki, hogy az első

feltételt kielégítő bármely elemekre a további feltételek is kielégíthetőek. A három kvantor valamelyikének hatáskörén belül megkötött Drools változó a hagyományos kvantor szemantikának megfelelően nem „látszik ki” a kvantoron kívülre.

Az objektumminta attribútum-korlátozásai formailag egy attribútumnévből állnak, amelyet egy korlátozás követ; itt is használható zárójelezés ill. `&&` vagy `||` operátor. A megnevezett attribútumokat a Drools a megfelelő getterek segítségével próbálja elérni; speciális esetként a **`this`** attribútumnév magára az objektumra vonatkozik. A korlátozások az attribútumot egy (akár Drools változókat is felhasználó) kifejezéshez viszonyítják. A hagyományos Java összehasonlító operátorok (pl. ==, <), mellett néhány Drools-specifikus kiegészítés (pl. regexp illesztés) is elérhető. A legfontosabb kiegészítések (**`memberOf`**, **`not memberOf`**, **`contains`**, **`not contains`**) a kollekciók elemvizsgálatát végzik.

A fenti példánál több Drools nyelvi elemet illusztrál az alábbi szabály, amely azt mondja ki, hogy a konyhapulton heverő üres szilvákba egy–egy új kockacukrot helyezünk.:

```
rule "Cukroz"
      when
            $szilva: Szilva ()
            $pult: KonyhaPult (tartalom contains $szilva)
            not Object (this memberOf $szilva.tartalom) ## ures a szilva
      then
            KockaCukor $cukor = new KockaCukor();
            insert($cukor);
            modify($szilva) {
                  berak($cukor);
            }
end
```

A feltételrészben az első feltétel egy `Szilva` osztályba tartozó objektumra vonatkozó minta, amely a `$szilva` változóban lesz megkötve. A második minta azt a konyhapultot köti meg egy változóba, amelyiknek a `getTartalom()` metódusa olyan kollekciót ad vissza, amelyik tartalmazza a szilvát. Itt érdemes megjegyezni, hogy abban az esetben is jó ez a konstrukció, ha csak egy konyhapult van, és nem minden szilva van konyhapulton; ekkor inkább a szilvára, mint a konyhapultra érződik megkötésnek ez az attribútum-korlátozás, de az eredmény természetesen logikailag ekvivalens. A harmadik feltétel azt mondja, hogy nincs olyan objektum a WM-ben, amelyik benne lenne a szilva tartalom-kollekciójában (vagyis a szilva üres, nincs már benne mag, de még cukor sem).

Az akció rész először megpéldányosít egy kockacukrot, majd a WM-be helyezi. Utána a `$szilva` objektum `berak()` metódusát hívja meg, a cukorral paraméterezve; egyúttal értesíti a Drools-t arról, hogy a szilva objektum attribútumai frissültek és újraolvasandók.

Nem beszéltünk még a szabály opcióiról, amelyeket a **`when`** kulcsszó előtt lehet megadni. Az egyik legfontosabb opció a **`salience`**, amelyet egy egész szám követ. A megadott szám lesz a szabály prioritása (alapértelmezésben 0), és több aktivált szabály esetén a legmagasabb prioritású fog tüzelni.

Egy másik fontos opció az alapértelmezésben hamis értékű **`no-loop`**. Volt már arról szó, hogy a szabálypéldány végrehajtásakor a saját aktivációja automatikusan eltűnik az agendáról, hogy azok a szabályok se kerüljenek „végtelen ciklusba”, amelyek nem érvénytelenítik saját előfeltételeiket. Ha azonban az érintett szabály módosítja a feltételrészében szereplő valamelyik objektumot, akkor a Drools újra észleli az aktiváció érvényességét és visszateszi az agendára. Ez kerülhető el **`no-loop true`** megadásával. Tartsuk észben, hogy a több szabályon keresztül lezajló kölcsönös rekurzió ellen ez sem hatásos; az igazi megoldás az, ha minden szabály valamilyen módon érvényteleníti saját aktivációját (pl. kockacukrot tesz a szilvába, amely így már nem lesz üres).

A nyelv részletesebb, pontosabb, példákban gazdagabb leírása az online dokumentációban olvasható.

## 2.6 Drools fejlesztőkörnyezet és hibakeresés

A Drools Eclipse-be épülő fejlesztői környezetet biztosít. A szabálydefiníciós fájlokhoz fejlett editor támogatás tartozik (pl. környezetérzékeny kódkiegészítés), ezen kívül projekt varázslóval létrehozhatóak Hello World jellegű projektek (a tudásbázist felépítő kódot innen célszerű eltanulni).

A fejlesztőkörnyezet legfontosabb feladata a hibakeresés támogatása. Ehhez a tevékenységhez nagyon fontos segítséget nyújt az Audit nézet (Window > Show View > Other… > Drools > Audit), ahol visszamenőleg áttekinthetőek a végrehajtott szabálypéldányok, az elvégzett WM módosítások, és a hatásukra megjelent vagy megszűnt aktivációk. Ehhez azonban be kell kapcsolni a loggolást, majd az Audit nézetet az elkészült log fájlra irányítani. A loggolás a session létrejötte után bekapcsolható a `KnowledgeRuntimeLoggerFactory.newFileLogger(session, "logFileNeve.log")` hívással. A visszakapott `KnowledgeRuntimeLogger` objektum a program befejeződésekor lezárandó (`close()`).

Ha töréspontnál (pl. a `fireAllRules()` hívás előtt, vagy egy Drools szabály RHS részében) megállítjuk a programunkat, akkor további információkat kaphatunk a pillanatnyi állapotról. Az Agenda nézet a conflict set tartalmát mutatja, a Working Memory pedig értelemszerűen a WM-beli tényeket. Ha Java töréspontnál állunk meg, akkor a Drools debug környezetnek nincs tudomása a futó Drools BRE példányról, ezért először a Java debuggolásnál megszokott Variables nézetben ki kell jelölni a Drools session, és csak ezután töltődnek fel a Drools debug nézetei tartalommal. Ha azonban egy szabály akció részében elhelyezett töréspontnál állunk meg, akkor azonnal használhatóak ezek a nézetek, és a Variables nézet a Drools változókat mutatja. A szabály RHS-ben elhelyezett töréspontok csak akkor működnek, ha a programot Java Application helyett Drools Applicationként indítjuk, debug módban.

# 3 A mérés elvégzése

A mérés során a hallgatóknak a korábbi méréseken megvalósított egyszerű, de elosztott üzleti logika implementáció helyett központosított, de deklaratív szabályokkal definiált üzleti logikát kell kialakítaniuk. Mivel a szabályalapú formalizmus nem kifejezetten vezérlési folyam ill. sorrendiség modellezésére való, de cserébe komplex, nagy kifejezőerejű deklaratív feltételek adhatóak meg, ezért megengedhető kisebb mértékű eltérés a korábban meghatározott munkafolyamathoz képest. Ennek megfelelően esetleg feloldhatóak bizonyos sorrendi megkötések, feltéve hogy a működés ettől még helyes marad. Ha ez nem elegendő, akkor például az adatmodell kibővítésével tárolható, hogy egy adott munkadarab hol tart a munkafolyamatban. Az üzleti objektumok és egymáshoz való viszonyuk abból a célból is bonyolultabbá tehető, hogy ki lehessen használni a Drools kifejezőerejét (pl. több objektum vizsgálata).

Fontos követelmény, hogy az üzleti logikát teljes egészében szabályok határozzák meg; az üzleti objektumok önmagukban csupán adatmodellként működjenek.

A következő feladatokat kell elvégezni:

- Java adatmodell osztályok készítése az üzleti objektumok reprezentálására. Kommunikációt, döntést, üzleti logikát ne tartalmazzanak! GUI integráció szempontjából szükséges elemek (pl. változás listener hívása) maradhatnak.
- Drools szabályok létrehozása az üzleti logika lépéseinek megvalósításához.
    - A korábbi munkafolyamat-modell minden lépéséből készüljön legalább egy külön Drools szabály.
    - A szabályok képesek legyenek tetszőlegesen sok egymás után vagy átlapolva érkező input fogadására, lekezelésére.
    - Az üzleti objektummodell úgy választandó meg, hogy legyen legalább egy olyan szabály, ahol az LHS több objektummintából áll.
    - A szabályok akció része lehetőleg legyen minél egyszerűbb.
- A sessiont tesztadatokkal feltöltő és a szabálymotort elindító keretprogram elkészítése.

- Grafikus vagy szöveges visszajelzés adása az adatmodell pillanatnyi állapotáról, minimum a program befejeződésekor.

A mérés után leadott jegyzőkönyvben szerepeljen a mérés elvégzéséhez szükséges lépések leírása megismételhető módon. Legyen benne a megvalósított üzleti logika specifikációjának rövid leírása. Külön ki kell emelni azokat a specifikációmódosításokat, amelyek ehhez a méréshez születtek, továbbá. az adatmodell nemtriviális vagy a specifikációból ki nem következtethető elemeit. Szerepeljen a jegyzőkönyvben az elkészült szabályok felépítésének általános elve, a különféle tervezői döntések (pl. bizonyos munkafolyamat-csomópontok megvalósítása) dokumentációja, és néhány reprezentatív kódrészlet a szabályok definíciójából. Tartalmazzon megfelelő mennyiségű képernyőképet és útmutatót ahhoz, hogy hogyan kell lefuttatni az elkészült programot.

# 4 Ellenőrző kérdések

A beugró kérdések kizárólag ebben a dokumentumban leírtakat fogják visszakérdezni, azonban a mérés sikeres elvégzéséhez célszerű lehet a Drools dokumentáció használata is.

1. Milyen előnye és hátránya lehet, ha egy információs rendszer üzleti logikáját szabályalapú rendszerrel valósítjuk meg?
2. Milyen részekből épül fel egy produkciós szabály?
3. Milyen alapvető különbségek vannak abban, ahogy egy logikai következtető és egy BRE végrehajtja a szabályokat?
4. BRE esetén mi a beszúrás és logikai beszúrás közötti különbség?
5. A BRE rendszerek tipikusan milyen támogatást adnak sok hasonló szabály tömör megadásához?
6. Mit nevezünk egy szabály aktivációjának? Mely elemekből épül fel az aktiváció Drools esetén?
7. Mi a conflict set (agenda, napirend)?
8. Mi az az inkrementális mintaillesztés? Mi az előnye, és milyen feltételt szab az üzleti objektumok kezelésére?
9. Mit jelent a láncolás? Előre vagy hátra láncolást végez a Drools? Meddig végez láncolt szabályalkalmazást egy BRE?
10. Milyen következményekkel jár, ha egy szabályalkalmazás nem érvényteleníti a saját aktivációját? Mi a teendő ilyen helyzetben?
11. Mi az állapotos és állapotmentes Drools session közötti különbség? Melyik lesz alkalmas a mérési feladat megvalósítására?
12. Mire való az Eclipse-es Drools környezet Audit, Agenda ill. Working Memory nézete?
13. Mutassa be röviden a Drools legalább 3 részprojektjét!