KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Automatische verificatie en kwaliteitscontrole van UML-diagrammen met FO(.)

Thomas Vochten

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdoptie Gedistribueerde systemen

Promotor:
Prof. dr. Marc Denecker

Assessor:
TBD

Begeleider:
Matthias van der Hallen

Academiejaar 2016 – 2017

Voorwoord

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Thomas Vochten

Inhoudsopgave

Samenvatting

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

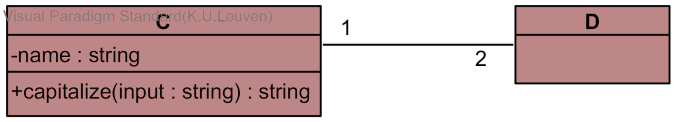
0.1 Inleiding

Binnen software engineering is UML een veelgebruikt gereedschap om het domein waarin de software die ontworpen wordt alsook de structuur van de software zelf grafisch weer te geven. Het is voor de ontwerper interessant om uit een tekstuele beschrijving van wat de voorgestelde software moet kunnen de relevante concepten en procedures te halen, die neer te zetten in een diagram en door middel van de verscheidene symbolen aangeboden door UML uit te drukken hoe die concepten en procedures met elkaar interageren. Op deze manier kan een team snel duidelijkheid scheppen in welke doelen ze precies moeten bereiken.

Het is echter makkelijk om het overzicht te verliezen als de gebruikte diagrammen omvangrijk worden. Dit kan een probleem zijn omdat fouten die worden gemaakt in de ontwerpfase en pas laat in het productieproces ontdekt worden kostbaar zijn om recht te zetten. Het komt ook voor dat een ontwerper per vergissing overbodige informatie toevoegt aan een diagram en dat daardoor het diagram minder duidelijk wordt.

In deze masterproef worden in het bijzonder UML-klassediagrammen beschouwd. Een klassediagram beschrijft welke concepten (in deze tekst verder *klasses* genoemd) er bestaan binnen de software. Elk van die klasses kan attributen en operaties hebben. Verder geeft een klassediagram ook weer welke klasses in relatie staan tot elkaar. Deze relaties leggen vast aan welke beperkingen alle mogelijke toestanden van de beschreven software moeten voldoen om beschouwd te worden als correct.

Beschouw volgend klassediagram:



FIGUUR 0.1: Een voorbeeld van een klassediagram

Dit klassediagram drukt uit dat er twee klasses bestaan: C en D. C heeft één attribuut, *name*, dat van type *string* is. Het heeft ook één operatie *capitalize* dat *input*, van type *string*, als parameter heeft. *capitalize* geeft een resultaat terug dat ook van type *string* is. Voorts drukt de lijn tussen C en D uit dat er een relatie bestaat tussen de twee klasses. Beschouw klasse C. Als we vanuit die klasse de lijn volgen, zien we dat er aan het ander uiteinde staat dat elke C-object in relatie moet staan tot exact twee D-objecten. Zo ook zien we dat, als we vertrekken vanuit D, elk D-object in relatie moet staan tot exact één C-object.

Met het voorgaande in het achterhoofd beschouwen we in deze masterproef twee categorieën van gebreken in een klassediagram:

- **Inconsistenties:** Het klassediagram is zo opgebouwd dat geen enkele mogelijke toestand van de software kan beantwoorden aan de voorwaarden die worden opgelegd. Dit betekent dat het stuk van de software dat wordt beschreven in het diagram onmogelijk kan werken.

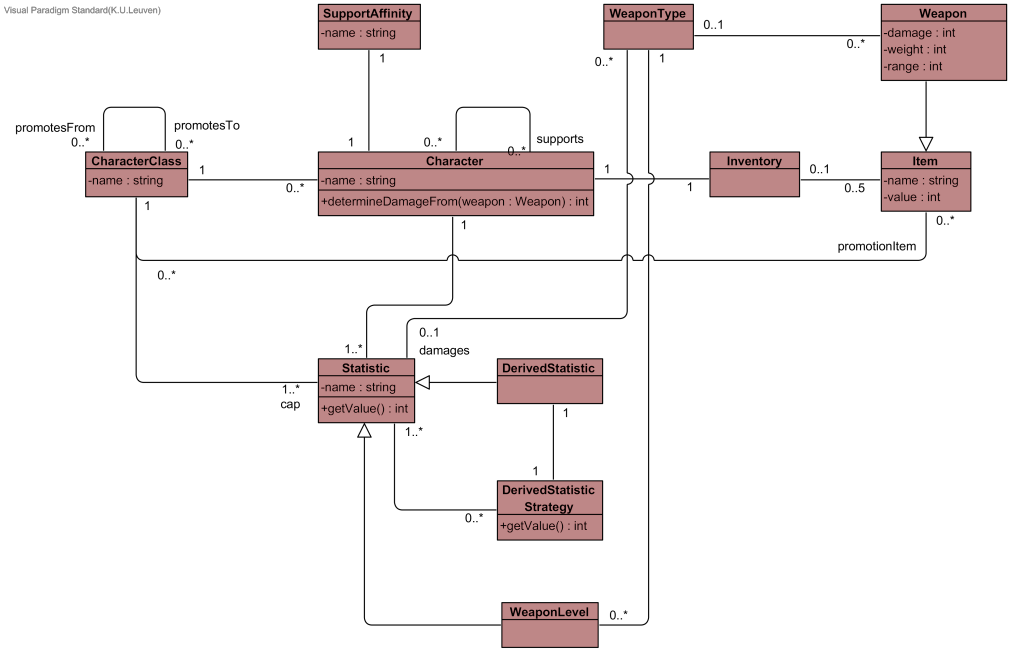
- **Kwaliteitsgebreken:** Deze gebreken hebben een negatieve impact op de kwaliteit van het softwareontwerp. Zo kunnen ze bijvoorbeeld onduidelijkheden in het ontwerp introduceren of het onderhoud van de software éénmaal ingezet in productie bemoeilijken.

Deze masterproef heeft tot doel om automatisch op een gestructureerde manier uit te drukken welke informatie een UML-klassediagram juist bevat. Die informatie willen we op zijn beurt terug gebruiken om inconsistenties en kwaliteitsgebreken te detecteren. Concreter willen we predikatenlogica gebruiken om de informatie neer te schrijven en om aan detectie van gebreken te doen. De volgende hoofdstukken beschrijven hoe we dit exact willen bereiken.

relevante structuur van document beschrijven

0.2 Controleren van consistentie

In dit hoofdstuk treden we meer in detail over hoe we de consistentie van een diagram willen controleren. Daarvoor willen we een specifieke vorm van logische theorie automatisch laten genereren. In deze theorieën staan **objecten** centraal. Deze objecten zijn instanties van een klasse die voorkomt in het beschouwde diagram, hebben exact de attributen en operaties van die klasse en maken deel uit van exact die relaties die het diagram voorschrijft voor die klasse. Aan de hand van volgend voorbeeld zullen we illustreren welke regels we gebruiken om zulk een theorie op te bouwen:



FIGUUR 0.2: Leidend voorbeeld van een klassediagram

Meer bepaald willen we uitdrukken welke **klasses** er bestaan in het diagram waarvan een object een instantie kan zijn, welke **attributen** en **operaties** elke klasse bevat, welke **associaties** er bestaan tussen de verscheidene klasses en welke **klassehiërarchieën** er bestaan.

0.2.1 Logisch type *Object*

Zoals gezegd staan in deze theoriën objecten centraal, dus is het vanzelfsprekend om een logisch type *Object* te voorzien. Dit logisch type bevat dus softwareobjecten.

0.2.2 Logisch type *ClassObject* en predicaat *StaticClass*\2

Dit logisch type bevat exact de klasses die worden weergegeven in het diagram — niet meer en niet minder. *Character*, *Inventory*, *Item* enz. zijn dus *ClassObject*s.

We drukken uit dat een *Object* een instantie is van een bepaald *ClassObject* door middel van het predicaat *StaticClass*\2. *StaticClass(o1,Character)* zegt dus uit dat het object *o1* een instantie is van klasse *Character*.

0.2.3 Voorstellen van attributen

Voor elk attribuut voegen we een binair predicaat toe waarvan de naam beantwoordt aan het patroon: *Klassenaamattribuutnaam*. Voor klasse *Character* en attribuut *name* resulteert dit dus in het predicaat *Charactername*\2. Het eerste argument van dit predicaat is een *Object*. Het type van het tweede argument hangt af van wat er in het diagram staat: Als het een primitief type is zoals *string* of *int*, zal dat ook het type zijn van het tweede predicaat; in het andere geval is het type van het tweede argument ook *Object*. De signatuur van *Charactername*\2 is daarom *Charactername(Object, string)*. Voor elk attribuut worden ook een aantal andere regels afgeleid:

- Als het tweede argument van het attribuutpredicaat *Object* is, wordt er een regel toegevoegd van de vorm:

$$\forall o1[Object]\forall o2[Object](Klassenaamattribuutnaam(o1, o2) \Rightarrow \\ StaticClass(classObj, o1) \wedge StaticClass(attrClassObj, o2))$$

  waarbij *classObj* het logisch object van type *ClassObject* dat het attribuut bevat en *attrClassObj* het logisch object van type *ClassObject* dat dient als mogelijke waarde van dit attribuut. Deze regel verzekert dat de attribuuthouder en de attribuutwaarde van de juiste klasse zijn. In dit diagram komt dit geval nergens voor en wordt deze regel dus niet toegepast.

- Als het tweede argument van het attribuutpredicaat van een primitief type is, wordt een regel toegevoegd van de vorm:

$$\forall o[Object]\forall x[primitiveType](Klassenaamattribuutnaam(o, x) \Rightarrow \\ StaticClass(classObj, o))$$

  waarbij primitiveType het type van de attribuutwaarde. De signatuur van het predicaat verzekert dat de attribuutwaarde van het juiste type is, dus moet dit niet explicit worden neergeschreven. Deze regel zorgt ervoor dat de volgende zin wordt toegevoegd aan de theorie:

$$\forall o[Object]\forall x[primitiveType](Charactername(o, x) \Rightarrow \\ StaticClass(Character, o))$$

- De multipliciteit van het attribuut wordt ook in rekening gebracht. Zij *lowerBound* de ondergrens en *upperBound* de bovengrens. Dan is de meest algemene vorm van deze regel als volgt:

$$\forall o1[Object](StaticClass(classObj, o1) \Rightarrow lowerBound \geq \\ \#\{o2 : Klasseattribuutnaam(o1, o2)\} \geq upperBound$$

  waarbij *lowerBound* wordt weggelaten als deze 0 is en *upperBound* wordt weggelaten als deze $*$ is. Indien beide van deze voorwaarden gelden, wordt er geen regel afgeleid betreffende de multipliciteit van het attribuut. Als $lowerBound = upperBound$, wordt deze regel in de plaats:

$$\forall o1[Object](StaticClass(classObj, o1) \Rightarrow \\ \exists_{=upperBound} o2(Klassenaamattribuutnaam(o1, o2))$$

  Voor *Charactername*\2 wordt daarom afgeleid:

$$\forall o[Object](StaticClass(Character, o) \Rightarrow \exists_{=1} x(Charactername(o, x))$$

0.2.4 Voorstellen van operaties

Voor elke operatie voegen we een predicaat toe dat beantwoordt aan volgend patroon: *Klassenaamoperatienaam*\$(m+2)$, waarbij m het aantal argumenten dat als invoer wordt meegegeven aan de operatie. De signatuur ziet eruit als $Klasseoperatienaam(o, p_1, \ldots, p_m, r)$, waarbij o het object van logisch type *Object* waarop de operatie wordt opgeroepen, $p_1 \ldots p_m$ de argumenten en r het resultaat van de oproep van de operatie op het object o met de gegeven argumenten. Indien er geen argumenten zijn, ziet de signatuur eruit als *Klassenaamoperatienaam(o,r)*. Voor *determineDamageWeaponFrom(Weapon)* van *Character* wordt dit dus *CharacterdetermineDamageFrom(Object,Object,int)*.

Voor elke operatie worden de volgende regels afgeleid:

- Het object waarop de operatie wordt opgeroepen (zijnde o), de parameters (zijnde $p_1 \ldots p_m$) en het resultaat van de oproep (zijnde r) moeten allemaal van de juiste klasse zijn. Daarom wordt een regel toegevoegd van de vorm:

$$\forall o[Object] \forall p_1[Object] \ldots \forall p_m[Object] \forall r[Object] \\ (Klassenaamoperatienaam(o, p_1, \ldots, p_m, r) \Rightarrow \\ StaticClass(classObj, o) \wedge StaticClass(p_1 ClassObj, p_1) \wedge \ldots \wedge \\ StaticClass(p_m ClassObj, p_m) \wedge StaticClass(resultClassObj, r))$$

Voor elke p waarvoor geldt dat het van een primitief type is wordt de corresponderende *StaticClass(p_l, p_lClassObj)* (met $1 \leq l \leq m$) weggelaten; hetzelfde geldt voor r. De invulling voor *CharacterdetermineDamageFrom(o,p1,r)* wordt dus:

$$\forall o[Object]\forall p_1[Object](CharacterdetermineDamageFrom(o, p_1, r) \Rightarrow \\ StaticClass(Character, o) \wedge StaticClass(Weapon, p_1))$$

- Voor elke combinatie van Object waar de operatie wordt opgeroepen en invoerparameters moet gelden dat er exact één resultaat is:

$$\forall o[Object]\forall p_1[Object] \ldots \forall p_m[Object] \\ (StaticClass(classObj, o) \wedge StaticClass(p_1ClassObj, p_1) \wedge \ldots \wedge \\ StaticClass(p_m, p_mClassObj) \Rightarrow \\ \exists! r[Object](Klassenaamoperatienaam(o, p_1, \ldots, p_m, r)))$$

Opnieuw geldt dat voor primitieve types de bijhorende conjuncten weggelaten worden. De invulling voor *CharacterdetermineDamageFrom(Object,Object,int)* wordt:

$$\forall o[Object]\forall p_1[Object](StaticClass(Character, o) \wedge \\ StaticClass(Weapon, p_1) \Rightarrow \exists! r(CharacterdetermineDamageFrom(o, p1, r))$$

0.2.5 Voorstellen van associaties

Voor elke associatie voegen we een predicaat toe dat beantwoordt aan volgend patroon: $ClassOneand \ldots andClassM \backslash m$, waarbij m de ariteit van de associatie. Voor de associatie tussen *Inventory* en *Item* wordt dit dus *InventoryandItem(Object,Object)*. We leiden regels van de volgende vormen af voor elke associatie:

- De deelnemende Objects moeten allemaal van de juiste klasse zijn. Daarom wordt een regel toegevoegd van de vorm:

$$\forall o_1[Object] \ldots \forall o_m[Object](ClassOne \ldots andClassM(o_1, \ldots, o_m) \\ \Rightarrow StaticClass(o_1ClassObj, o_1) \wedge \ldots \wedge StaticClass(o_mClassObj, o_m))$$

Voor *InventoryandItem(Object,Object)* wordt dit:

$$\forall o_1[Object]\forall o_2[Object](InventoryandItem(o1, o2) \Rightarrow \\ StaticClass(Inventory, o1) \wedge StaticClass(Item, o2))$$

- De multipliciteit voor elke rol moet worden uitgedrukt. Voor alle o_l waarvoor $1 \leq l \leq m$ wordt een regel toegevoegd van de volgende vorm:
  Zij $lowerBound_l$ de ondergrens en $upperBound_l$ de bovengrens:

  $$\forall c_1[Object] \ldots \forall c_m[Object](StaticClass(c_1 ClassObj, c_1) \wedge \ldots \wedge$$
  $$StaticClass(c_m ClassObj, c_m) \Rightarrow lowerBound_l \leq$$
  $$\#o_l : ClassOneand \ldots ClassM(c_1, \ldots, o_1, \ldots, c_m) \leq upperBound_l)$$

  waarbij de c met index l overgeslagen wordt. Indien de ondergrens gelijk is aan 0 of de bovengrens gelijk is aan $*$ worden dezen weggelaten. Als beide voorwaarden gelden, wordt voor deze l geen regel afgeleid. Indien $lowerBound_l = upperBound_l$ wordt in de plaats afgeleid:

  $$\forall c_1[Object] \ldots \forall c_m[Object](StaticClass(c_1 ClassObj, c_1) \wedge \ldots \wedge$$
  $$StaticClass(c_m ClassObj, c_m) \Rightarrow$$
  $$\exists_{=upperbound_l} o_l(ClassOneand \ldots andClassM(c_1, \ldots, o_l, \ldots, c_m)))$$

  Voor $InventoryandItem\backslash m$ worden de volgende regels afgeleid:

  $$\forall o_2[Object](StaticClass(Item, o_2) \Rightarrow$$
  $$\#o_1 : InventoryandItem(o_1, o_2) \leq 1)$$

  $$\forall o_1[Object](StaticClass(Inventory, o_1) \Rightarrow$$
  $$\#o_2 : InventoryandItem(o_1, o_2) \leq 5)$$

0.2.6 Voorstellen van klassehiërarchiëen

Stel dat voor een object o van logisch type *Object* gegeven is dat *StaticClass(oClassObject,o)*. Ons doel is dat *StaticClass(superClassObject,o)* geldt voor alle objecten van logisch type *ClassObject* die volgens het diagram superklasses zijn van *oClassObject* — niet meer en niet minder. Daartoe introduceren we het predikaat *IsDirectSupertypeOf(ClassObject,ClassObject)* dat ingevuld wordt door alle directe subklasseringen van het diagram en het predikaat *IsSupertypeOf(ClassObject,ClassObject)*, hetgeen de transitieve sluiting is van *IsDirectSupertypeOf*. Om zowel de invulling van *IsDirectSupertypeOf*$\backslash 2$ te doen als de transitieve sluiting te berekenen maken we gebruik van **inductieve definities** voor twee redenen:

1. In predikatenlogica is het onmogelijk om op een universeel geldige manier de transitieve sluiting uit te drukken.
2. Als men in een inductieve definitie een lijst feiten opsomt, drukt men tegelijk ook uit dat exact die feiten waar zijn — niet meer of niet minder.

In één definitie lijsten we dus de feiten die we kunnen aflezen van het diagram op:

$$\begin{aligned}\{IsDirectSupertypeOf(Statistic, Weaponlevel) \leftarrow \\ IsDirectSupertypeOf(Statistic, DerivedStatistic) \leftarrow \\ IsDirectSupertypeOf(Item, Weapon) \leftarrow \\ \}\end{aligned}$$

In een andere definitie drukken we de transitieve sluiting uit en gebruiken we die ook meteen om het gewenste resultaat voor *StaticClass*\2 uit te komen:

$$\begin{aligned}&\{\forall x[ClassObject]\forall y[ClassObject](IsSupertypeOf(x,y) \leftarrow \\ &\ IsDirectSupertypeOf(x,y)) \\ &\ \forall x[ClassObject]\forall y[ClassObject](IsSupertypeOf(y,x) \leftarrow \\ &\ \exists z(IsSupertypeOf(y,z) \wedge IsSupertypeOf(z,x)))\end{aligned}$$

$$\begin{aligned}&\forall x[ClassObject]\forall o[Object](StaticClass(x,o) \leftarrow RuntimeClass(x,o)) \\ &\forall x[ClassObject]\forall y[ClassObject]\forall o[Object](StaticClass(y,o) \leftarrow \\ &RuntimeClass(x,o) \wedge IsSupertypeOf(y,x))\}\end{aligned}$$

waarbij *RuntimeClass(ClassObject,Object)* een predikaat is dat uitdrukt wat de uniek dynamisch bepaalde klasse is van een *Object* (de veronderstelling is dat in een geldige toestand van een programma in uitvoering ieder object exact één *runtime* klasse heeft).
In hoofdstuk 0.4 wordt de logische theorie die automatisch gegenereerd werd volgens de regels opgelijst in dit hoofdstuk weergegeven en wordt ook uitgelegd hoe die theorie wordt gebruikt om de consistentie van het diagram te controleren.

0.3 Controleren op kwaliteitsgebreken

Waar in hoofdstuk 0.2 *Object*s centraal stonden, doen we daar hier afstand van: we abstraheren *Object*s weg en concentreren ons in de plaats op *ClassObject*s. We gebruiken het diagram in figuur 0.2 weer als begeleidend voorbeeld. In de volgende subsecties overlopen we hoe we de theorie die we gebruiken voor dit probleem opbouwen.

0.3.1 Gebruikte logische types en predikaten

We bewaren het logisch type *ClassObject* en het predikaat *IsSupertypeOf(ClassObject, ClassObject)* exact zoals ze zijn in hoofdstuk 0.2 (en berekenen de transitieve sluiting horende bij *IsSupertypeOf*\2 op dezelfde manier) en gebruiken daar bijkomend volgende predikaten:

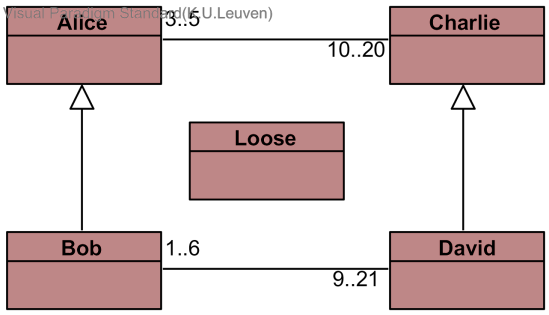
- ***BiAssoc(ClassObject,ClassObject)***: drukt uit dat er een binaire associatie bestaat tussen de twee klasses.
- ***BiAssocLow(ClassObject,ClassObject,ClassObject,nat)***: voor *BiAssocLow(x,y,x,n1)* geldt dat voor de binaire associatie tussen klasse x en klasse y de ondergrens voor de multipliciteit aan de x-kant gelijk is aan *n1*; een gelijkaardige interpretatie geldt voor *BiAssocLow(x,y,y,n2)*.
- ***BiAssocHigh(ClassObject,ClassObject,ClassObject,nat)***: gelijkaardig aan *BiAssocLow*\4, maar dan voor de bovengrens van de multipliciteit.

Deze predikaten worden ingevuld met een lijst van feiten die af te lezen zijn van het diagram. In hoofdstuk 0.4 wordt de logische theorie die het resultaat is van dit proces weergegeven.

0.3.2 Kwaliteitsgebreken detecteren

Er zijn drie kwaliteitsgebreken waarnaar wordt gezocht in de resulterende theorie:

- **Many-to-many associaties**: Dit zijn associaties waar dat de bovengrens van de multipliciteiten aan beide kanten gelijk is aan $*$. Het voorkomen van een many-to-many associatie is doorgaans een teken dat er een klasse ontbreekt in het ontwerp. Het is dus van groot belang dat dit wordt opgespoord en opgelost.
- **Losstaande klasse**: Concreet is een losstaande klasse een klasse die geen associatie heeft met een andere klasse in het ontwerp. Zulk een klasse is nutteloos en moet ofwel verbonden worden met een andere klasse of verwijderd worden.
- **Overbodige associaties in een klassehiërarchie**: Beschouw figuur 0.3.2. Daar is te zien dat de grenzen van de multipliciteiten in de associatie *Alice—Charlie* strengere voorwaarden opleggen dan die van de associatie *Bob—David*,



FIGUUR 0.3: Voorbeeld van meer permissieve multipliciteiten in een klassehiërarchie en van een losstaande klasse (zijnde *Loose*)

en dat daarom de associatie *Bob*—*David* overbodig is. Om de verstaanbaarheid van het diagram te verbeteren, wordt die associatie best verwijderd.

We definiëren deze respectievelijke gebreken in de logische theorie door middel van de volgende logische zinnen:

$$\forall x[ClassObject]\forall y[ClassObject](ManyToMany(x,y) \Leftrightarrow BiAssoc(x,y) \wedge \neg\exists z[nat](BiAssocHigh(x,y,x,z)) \wedge \neg\exists z[nat](BiAssocHigh(x,y,y,z)))$$

$$\forall x[ClassObject](LooseClass(x) \Leftrightarrow \neg(\exists y[ClassObject](BiAssoc(x,y)) \vee \exists s[ClassObject]\exists y[ClassObject](IsSupertypeOf(s,y) \wedge BiAssoc(s,x))))$$

oplossing vinden voor de derde: gewoon verwijzen naar IDP-bestand?

Deze regels worden meteen toegevoegd aan de theorie die wordt gegenereerd zoals uitgelijnd eerder in dit hoofdstuk. In hoofdstuk 0.4 wordt uitgelegd hoe deze theorie wordt gebruik om kwaliteitsgebreken te vinden.

0.4 De rol van IDP

0.4.1 Korte inleiding in IDP

IDP is een kennisbanksysteem. Deze kennisbanken zijn opgesteld in FO(·), een uitbreiding van predikatenlogica. De basisblokken van een specificatie in IDP zijn als volgt:

- **Vocabularium**: Hier specificeert de ontwerper de logische types die bestaan in het beschouwd domein, predikaten en functies
- **Theorie**: Hier schrijft de ontwerper zinnen in FO(·) die bepalen welke structuren over het beschouwde vocabularium modellen zijn (waarbij natuurlijk niet wordt uitgesloten dat de ontwerper een inconsistente theorie ontwerpt).
- **Structuur**: De ontwerper vult hier de logische types gedefinieerd in het vocabularium in (waar nodig) en geeft voor één of meerdere predikaten aan welke tupels wel of geen lid zijn, als hij dat wil.

De ontwerper kan meerdere vocabularia, theorieën en structuren neerschrijven. Elke theorie kan wel maar de symbolen van één vocabularium gebruiken en men kan in een structuur alleen spreken over één theorie.

IDP gebruikt zijn eigen symbolen voor universele kwantoren, existentiële kwantoren en logische connectieven. Voor meer informatie over IDP, zie

verwijzing naar relevant document

0.4.2 Gebruik van modeluitbreiding

Gegeven een structuur over een bepaalde theorie en vocabularium kan de gebruiker de opdracht geven aan IDP om een uitbreiding te vinden van deze structuur die ervoor zorgt de structuur een model is van de theorie. Dit is een vorm van inferentie die men **modeluitbreiding** noemt. Het kan echter het geval zijn dat IDP antwoordt dat zulk een uitbreiding niet bestaat of dat de uitvoering nooit eindigt.

Controleren van consistentie

In codebestand 0.4.3 staat de logische theorie die werd gegenereerd volgens de regels uitgelijnd in hoofdstuk 0.2. Als men dit geeft als invoer aan IDP, is het besluit dat er een model bestaat voor de theorie en dat het diagram inderdaad consistent is.

0.4.3 Detecteren van kwaliteitsgebreken

In codebestand 0.4.3 staat de logische theorie die werd gegenereerd uit een combinatie van de diagrammen uit figuren 0.2 en 0.3.2 volgens de regels uitgelijnd in hoofdstuk 0.3. IDP vindt alle many-to-many associaties, besluit dat *Loose* een losstaande klasse is en dat de associatie *B*—overbodig is door de samenloop van de klassehiërarchie en de opgelegde multipliciteiten.

kuis codebestanden flink op

```
vocabulary V {
    type LimitedInt = { 1..18 } isa int
    type LimitedFloat = { 0.0; 0.5; 1.0; 1.5; 2.0; 2.5; 3.0; 3.5; 4.0; 4.5;
        5.0; 5.5; 6.0; 6.5; 7.0; 7.5; 8.0; 8.5 } isa float
    type LimitedString  = { "SVObSVpOwNsLeM1TCYkx"; "s0mJOUkvFuOYxoypf0e2"; "
        ucPRTRrfWBdnx8IbebrH"; "IIsx8mkhNB6tFKXhIh01"; "c8FoPQm8gzGloJi352R6";
         "QOwcTPcuxqohdJ0OoYI5"; "nNhaMFI1sNq4FM9g9PFK"; "THmoxPvd1k7axZ9Rx3Vo
        "; "psOJIEnr6CUFa2S1shdP"; "2ykrKTZkDAopHEMGzBgp"; "
        YrUOvIjCy2OZLs39PE5t"; "LXDiF7O705qElFzEF3WJ"; "NW3yPaSUa5NJERB5bpd0";
         "NzD42F9XGUvbUNaZHU0q"; "s9Iudo9RU7iwdNeSJi8t"; "iNl5Hkr0r9krSOlg2KER
        "; "bkl5G0Ix9OUrFJfV1H7P"; "kC4BfZXQ4VDHxUmJ1O5G" } isa string
    type bool constructed from { true, false }
    type void constructed from { void }
    type Object
    type ClassObject constructed from { DerivedStatisticStrategy,
        DerivedStatistic, WeaponType, CharacterClass, Inventory, Weapon,
        Character, SupportAffinity, WeaponLevel, Statistic, Item }
    RuntimeClass(ClassObject, Object)
    StaticClass(ClassObject, Object)
    IsDirectSupertypeOf(ClassObject, ClassObject)
    IsSupertypeOf(ClassObject, ClassObject)

    CharacterClassname(Object,LimitedString)
    Weapondamage(Object,LimitedInt)
    Weaponweight(Object,LimitedInt)
    Weaponrange(Object,LimitedInt)
    Charactername(Object,LimitedString)
    SupportAffinityname(Object,LimitedString)
    Statisticname(Object,LimitedString)
    Itemvalue(Object,LimitedInt)
    Itemname(Object,LimitedString)


    DerivedStatisticStrategygetValue(Object,LimitedInt)
    CharacterdetermineDamageFrom(Object,Object,LimitedInt)
    StatisticgetValue(Object,LimitedInt)


    StatisticandCharacterClass(Object,Object)
    DerivedStatisticStrategyandStatistic(Object,Object)
    WeaponTypeandWeaponLevel(Object,Object)
    CharacterandStatistic(Object,Object)
    CharacterandInventory(Object,Object)
    CharacterandCharacter(Object,Object)
    WeaponTypeandWeapon(Object,Object)
    DerivedStatisticStrategyandDerivedStatistic(Object,Object)
    StatisticandWeaponType(Object,Object)
    ItemandInventory(Object,Object)
    ItemandCharacterClass(Object,Object)
    CharacterandCharacterClass(Object,Object)
    CharacterClassandCharacterClass(Object,Object)
    CharacterandSupportAffinity(Object,Object)

}

theory T:V {
    {
        ! x y : IsDirectSupertypeOf(x, y) <- x = Statistic & y = WeaponLevel.
        ! x y : IsDirectSupertypeOf(x, y) <- x = Item & y = Weapon.
        ! x y : IsDirectSupertypeOf(x, y) <- x = Statistic & y =
            DerivedStatistic.
    }

```

```
54      ! o : ?1 x : RuntimeClass(x, o).
55
56      {
57          ! x y : IsSupertypeOf(x, y) <- IsDirectSupertypeOf(x, y).
58          ! x y : IsSupertypeOf(y, x) <- ? z : IsSupertypeOf(y, z) &
                IsSupertypeOf(z, x).
59
60          ! x o : StaticClass(x, o) <- RuntimeClass(x, o).
61          ! x y o: StaticClass(y, o) <- RuntimeClass(x, o) & IsSupertypeOf(y, x)
                .
62      }
63
64      ! o x : CharacterClassname(o, x) => StaticClass(CharacterClass, o).
65      ! o : StaticClass(CharacterClass, o) => ?1 x : CharacterClassname(o, x).
66
67      ! o x : Weapondamage(o, x) => StaticClass(Weapon, o).
68      ! o : StaticClass(Weapon, o) => ?1 x : Weapondamage(o, x).
69
70      ! o x : Weaponweight(o, x) => StaticClass(Weapon, o).
71      ! o : StaticClass(Weapon, o) => ?1 x : Weaponweight(o, x).
72
73      ! o x : Weaponrange(o, x) => StaticClass(Weapon, o).
74      ! o : StaticClass(Weapon, o) => ?1 x : Weaponrange(o, x).
75
76      ! o x : Charactername(o, x) => StaticClass(Character, o).
77      ! o : StaticClass(Character, o) => ?1 x : Charactername(o, x).
78
79      ! o x : SupportAffinityname(o, x) => StaticClass(SupportAffinity, o).
80      ! o : StaticClass(SupportAffinity, o) => ?1 x : SupportAffinityname(o, x).
81
82      ! o x : Statisticname(o, x) => StaticClass(Statistic, o).
83      ! o : StaticClass(Statistic, o) => ?1 x : Statisticname(o, x).
84
85      ! o x : Itemvalue(o, x) => StaticClass(Item, o).
86      ! o : StaticClass(Item, o) => ?1 x : Itemvalue(o, x).
87
88      ! o x : Itemname(o, x) => StaticClass(Item, o).
89      ! o : StaticClass(Item, o) => ?1 x : Itemname(o, x).
90
91      ! o r : DerivedStatisticStrategygetValue(o, r) => (StaticClass(
            DerivedStatisticStrategy, o)).
92      ! o : (StaticClass(DerivedStatisticStrategy, o)) => (?1 r :
            DerivedStatisticStrategygetValue(o, r)).
93
94      ! o p1 r : CharacterdetermineDamageFrom(o, p1, r) => (StaticClass(
            Character, o) & (StaticClass(Weapon, p1))).
95      ! o p1 : (StaticClass(Character, o) & (StaticClass(Weapon, p1))) => (?1 r
            : CharacterdetermineDamageFrom(o, p1, r)).
96
97      ! o r : StatisticgetValue(o, r) => (StaticClass(Statistic, o)).
98      ! o : (StaticClass(Statistic, o)) => (?1 r : StatisticgetValue(o, r)).
99
100
101     ! o1 o2 : StatisticandCharacterClass(o1,o2) => ((StaticClass(Statistic, o1
            )) & (StaticClass(CharacterClass, o2))).
102     ! o2 : ((StaticClass(CharacterClass, o2))) => (1 =< #{o1 :
            StatisticandCharacterClass(o1,o2)}).
103     ! o1 : ((StaticClass(Statistic, o1))) => ?1 o2 :
            StatisticandCharacterClass(o1,o2).
104
105     ! o1 o2 : DerivedStatisticStrategyandStatistic(o1,o2) => ((StaticClass(
            DerivedStatisticStrategy, o1)) & (StaticClass(Statistic, o2))).
106     ! o1 : ((StaticClass(DerivedStatisticStrategy, o1))) => (1 =< #{o2 :
```

```
        DerivedStatisticStrategyandStatistic(o1,o2)}).
107
108     ! o1 o2 : WeaponTypeandWeaponLevel(o1,o2) => ((StaticClass(WeaponType, o1)
            ) & (StaticClass(WeaponLevel, o2))).
109     ! o2 : ((StaticClass(WeaponLevel, o2))) => ?1 o1 :
            WeaponTypeandWeaponLevel(o1,o2).
110
111     ! o1 o2 : CharacterandStatistic(o1,o2) => ((StaticClass(Character, o1)) &
            (StaticClass(Statistic, o2))).
112     ! o2 : ((StaticClass(Statistic, o2))) => ?1 o1 : CharacterandStatistic(o1,
            o2).
113     ! o1 : ((StaticClass(Character, o1))) => (1 =< #{o2 :
            CharacterandStatistic(o1,o2)}).
114
115     ! o1 o2 : CharacterandInventory(o1,o2) => ((StaticClass(Character, o1)) &
            (StaticClass(Inventory, o2))).
116     ! o2 : ((StaticClass(Inventory, o2))) => ?1 o1 : CharacterandInventory(o1,
            o2).
117     ! o1 : ((StaticClass(Character, o1))) => ?1 o2 : CharacterandInventory(o1,
            o2).
118
119     ! o1 o2 : CharacterandCharacter(o1,o2) => ((StaticClass(Character, o1)) &
            (StaticClass(Character, o2))).
120
121     ! o1 o2 : WeaponTypeandWeapon(o1,o2) => ((StaticClass(WeaponType, o1)) & (
            StaticClass(Weapon, o2))).
122     ! o2 : ((StaticClass(Weapon, o2))) => (#{o1 : WeaponTypeandWeapon(o1,o2)}
            =< 1).
123
124     ! o1 o2 : DerivedStatisticStrategyandDerivedStatistic(o1,o2) => ((
            StaticClass(DerivedStatisticStrategy, o1)) & (StaticClass(
            DerivedStatistic, o2))).
125     ! o2 : ((StaticClass(DerivedStatistic, o2))) => ?1 o1 :
            DerivedStatisticStrategyandDerivedStatistic(o1,o2).
126     ! o1 : ((StaticClass(DerivedStatisticStrategy, o1))) => ?1 o2 :
            DerivedStatisticStrategyandDerivedStatistic(o1,o2).
127
128     ! o1 o2 : StatisticandWeaponType(o1,o2) => ((StaticClass(Statistic, o1)) &
             (StaticClass(WeaponType, o2))).
129     ! o2 : ((StaticClass(WeaponType, o2))) => (#{o1 : StatisticandWeaponType(
            o1,o2)} =< 1).
130
131     ! o1 o2 : ItemandInventory(o1,o2) => ((StaticClass(Item, o1)) & (
            StaticClass(Inventory, o2))).
132     ! o2 : ((StaticClass(Inventory, o2))) => (#{o1 : ItemandInventory(o1,o2)}
            =< 5).
133     ! o1 : ((StaticClass(Item, o1))) => (#{o2 : ItemandInventory(o1,o2)} =< 1)
            .
134
135     ! o1 o2 : ItemandCharacterClass(o1,o2) => ((StaticClass(Item, o1)) & (
            StaticClass(CharacterClass, o2))).
136
137     ! o1 o2 : CharacterandCharacterClass(o1,o2) => ((StaticClass(Character, o1
            )) & (StaticClass(CharacterClass, o2))).
138     ! o1 : ((StaticClass(Character, o1))) => ?1 o2 :
            CharacterandCharacterClass(o1,o2).
139
140     ! o1 o2 : CharacterClassandCharacterClass(o1,o2) => ((StaticClass(
            CharacterClass, o1)) & (StaticClass(CharacterClass, o2))).
141
142     ! o1 o2 : CharacterandSupportAffinity(o1,o2) => ((StaticClass(Character,
            o1)) & (StaticClass(SupportAffinity, o2))).
143     ! o2 : ((StaticClass(SupportAffinity, o2))) => ?1 o1 :
```

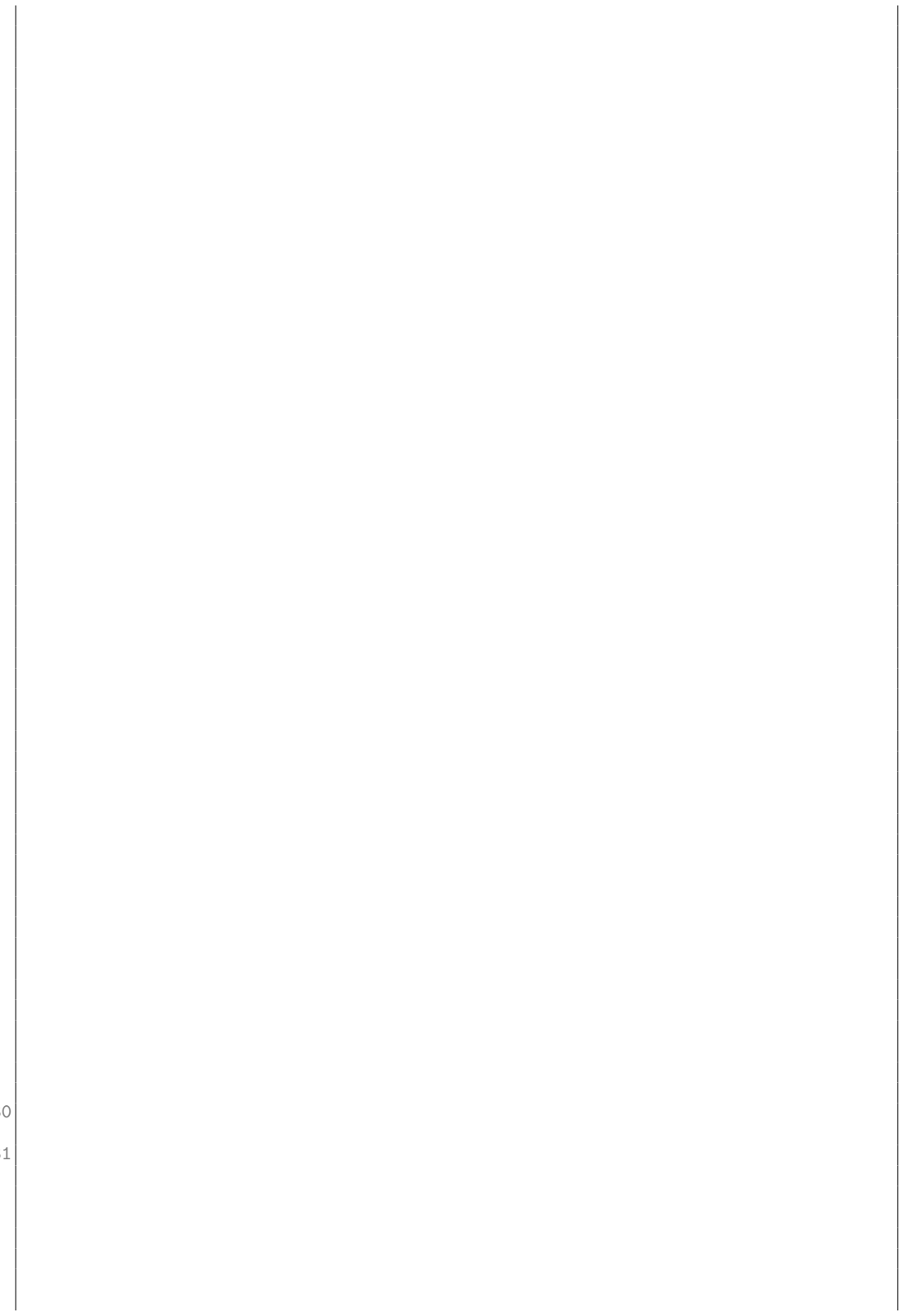
```
        CharacterandSupportAffinity(o1,o2).
144     ! o1 : ((StaticClass(Character, o1))) => ?1 o2 :
        CharacterandSupportAffinity(o1,o2).
145 }
146
147 structure thestruct : V {
148     Object = { 1..18}
149 }
150
151 procedure main() {
152     print(modelexpand(T,thestruct)[1])
153 }
```

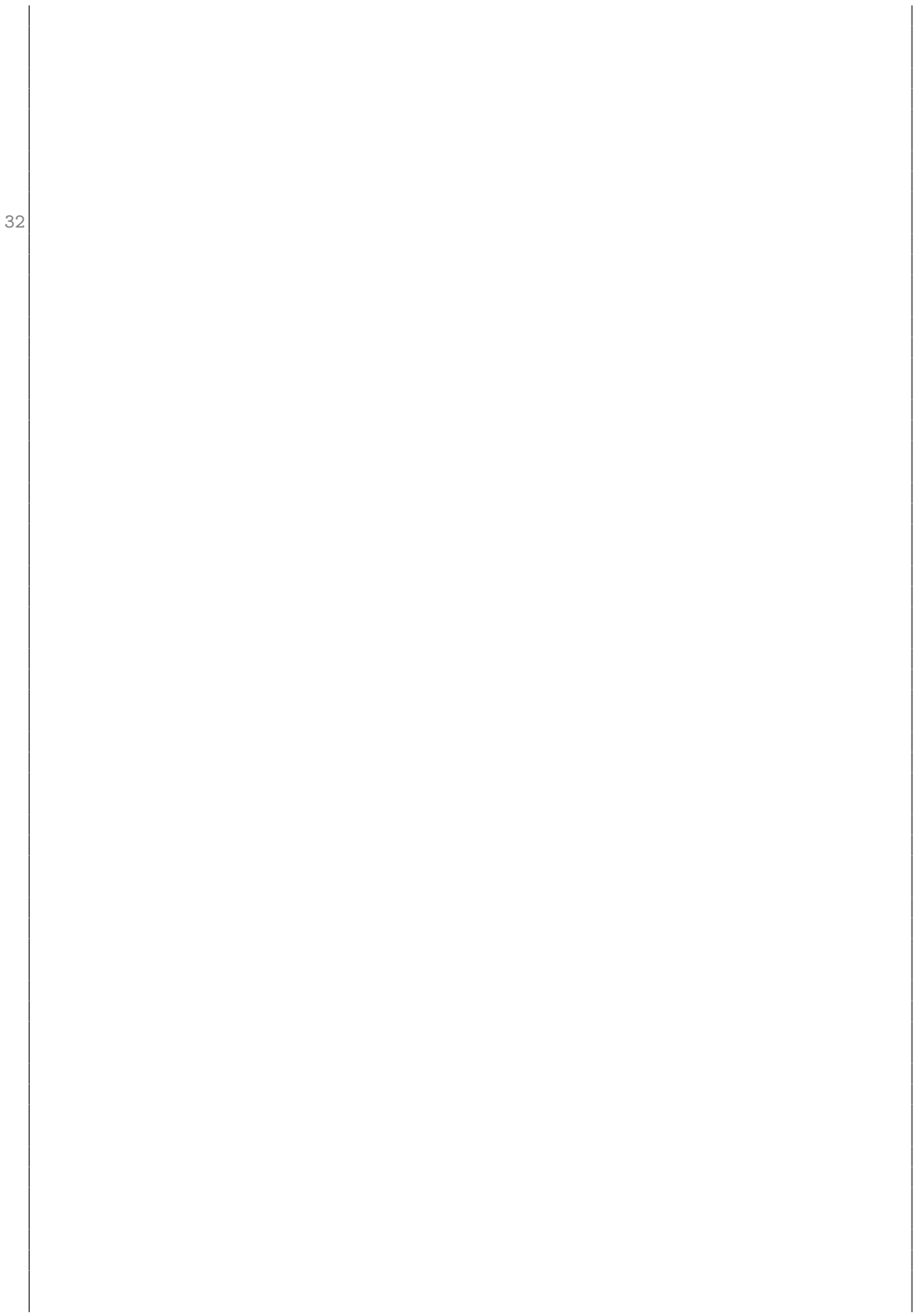
Codebestand 0.4.3

```
 1 vocabulary V {
 2     type ClassObject constructed from { DerivedStatisticStrategy,
           DerivedStatistic, WeaponType, CharacterClass, Inventory, Weapon,
           Character, SupportAffinity, WeaponLevel, Statistic, Item, Loose, A, B,
            C, D}
 3     type PrimitiveType constructed from { boolean, byte, character, double,
           floating, integer, long, short, astring, void}
 4     IsSupertypeOf(ClassObject, ClassObject)
 5
 6     type LimitedInt = {1 .. 21} isa int
 7
 8     BiAssoc(ClassObject, ClassObject)
 9     BiAssocLow(ClassObject, ClassObject, ClassObject, LimitedInt)
10     BiAssocHigh(ClassObject, ClassObject, ClassObject, LimitedInt)
11
12     ManyToMany(ClassObject, ClassObject)
13     LooseClass(ClassObject)
14     SubclassMorePermissiveMult(ClassObject, ClassObject, ClassObject)
15 }
16
17 theory T:V {
18
19     // ------- BAD DESIGN THAT MAY OCCUR IN UML -------
20
21     // many-to-many associations
22     ! x [ClassObject] y [ClassObject] : ManyToMany(x, y) <=> (BiAssoc(x, y) &
           ~ (? z [LimitedInt] : BiAssocHigh(x, y, x, z)) & ~ (? z [LimitedInt] :
            BiAssocHigh(x, y, y, z))).
23
24     // classes that are not associated with any other class
25     ! x [ClassObject] : LooseClass(x) <=> ~ ((? y [ClassObject] : BiAssoc(x, y
           )) | (? s [ClassObject] y [ClassObject] : IsSupertypeOf(s, x) & (
           BiAssoc(s, y)))).
26
27     ! x [ClassObject] y [ClassObject] : SubclassMorePermissiveMult(x, y, x)
           <=> ((? sx [ClassObject] : IsSupertypeOf(sx, x)
28                                                                    &
                                                                        ((
                                                                        BiAssoc
                                                                        (
                                                                        sx
                                                                        ,
                                                                        y
                                                                        )
```

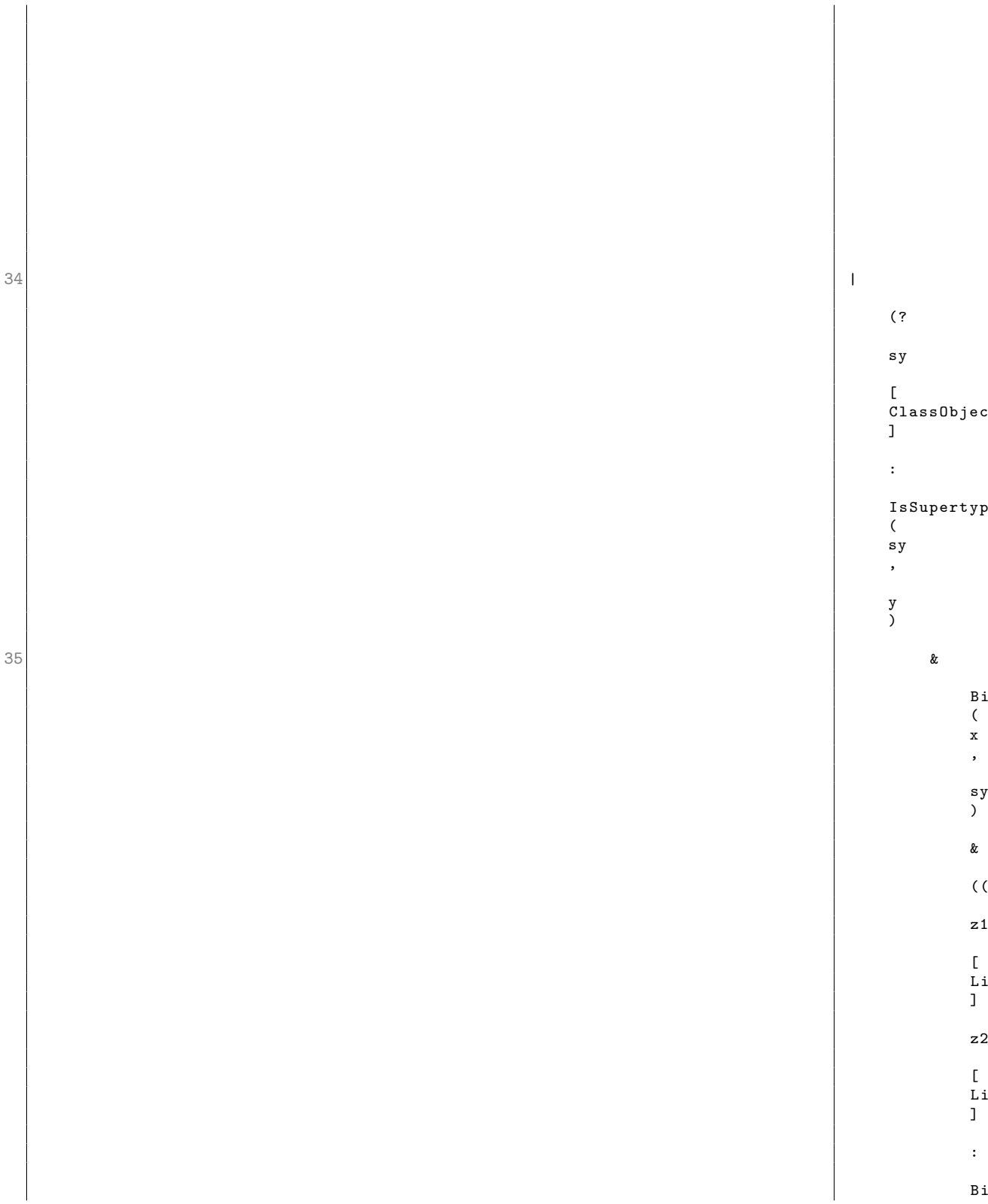
```
&
((?
z1
[
Limite
]
z2
[
Limite
]
:
BiAsso
(
x
,
y
,
x
,
z1
)
&
BiAsso
(
sx
,
y
,
sx
,
z2
)
&
z1
<
z2
)
```

29


30
31
(?
sy
[
ClassObjec
]



```
sx
,
z2
)
&
z1
<
z2
)
```


34
(?
sy
[
ClassObjec
]
:
IsSupertyp
(
sy
,
y
)
35
&
Bi
(
x
,
sy
)
&
((
z1
[
Li
]
z2
[
Li
]
:
Bi

```
(
x
,

y
,

x
,

z1
)

&

BiAssocLow
(
x
,

sy
,

x
,

z2
)

&

z1

<

z2
)
```

```
36
```

```
37      // ------- INFORMATION FROM DIAGRAM -------
38
39      // class hierarchy
40      {
41          IsSupertypeOf(Statistic, WeaponLevel) <- .
42          IsSupertypeOf(Item, Weapon) <- .
43          IsSupertypeOf(Statistic, DerivedStatistic) <- .
44          IsSupertypeOf(A,B) <- .
45          IsSupertypeOf(C,D) <- .
46          IsSupertypeOf(A,E) <- .
47          IsSupertypeOf(C,F) <- .
48      }
49
50      // associations between classes
51      {
52          BiAssoc(A, C) <- .
53          BiAssocLow(A, C, A, 3) <- .
54          BiAssocHigh(A, C, A, 5) <- .
55          BiAssocLow(A, C, C, 10) <- .
56          BiAssocHigh(A, C, C, 20) <- .
57
58          BiAssoc(B, D) <- .
59          BiAssocLow(B, D, B, 1) <- .
60          BiAssocHigh(B, D, B, 6) <- .
61          BiAssocLow(B, D, D, 9) <- .
62          BiAssocHigh(B, D, D, 21) <- .
63
```

```
64          BiAssoc(Statistic, CharacterClass) <- .
65          BiAssocLow(Statistic, CharacterClass, Statistic, 1) <- .
66          BiAssocLow(Statistic, CharacterClass, CharacterClass, 1) <- .
67          BiAssocHigh(Statistic, CharacterClass, CharacterClass, 1) <- .
68
69          BiAssoc(DerivedStatisticStrategy, Statistic) <- .
70          BiAssocLow(DerivedStatisticStrategy, Statistic, Statistic, 1) <- .
71          BiAssoc(WeaponType, WeaponLevel) <- .
72
73          BiAssocLow(WeaponType, WeaponLevel, WeaponType, 1) <- .
74          BiAssocHigh(WeaponType, WeaponLevel, WeaponType, 1) <- .
75
76          BiAssoc(Character, Statistic) <- .
77          BiAssocLow(Character, Statistic, Character, 1) <- .
78          BiAssocHigh(Character, Statistic, Character, 1) <- .
79          BiAssocLow(Character, Statistic, Statistic, 1) <- .
80
81          BiAssoc(Character, Inventory) <- .
82          BiAssocLow(Character, Inventory, Character, 1) <- .
83          BiAssocHigh(Character, Inventory, Character, 1) <- .
84          BiAssocLow(Character, Inventory, Inventory, 1) <- .
85          BiAssocHigh(Character, Inventory, Inventory, 1) <- .
86
87          BiAssoc(Character, Character) <- .
88
89          BiAssoc(WeaponType, Weapon) <- .
90          BiAssocHigh(WeaponType, Weapon, WeaponType, 1) <- .
91
92          BiAssoc(DerivedStatistic, DerivedStatisticStrategy) <- .
93          BiAssocLow(DerivedStatistic, DerivedStatisticStrategy,
                DerivedStatistic, 1) <- .
94          BiAssocHigh(DerivedStatistic, DerivedStatisticStrategy,
                DerivedStatistic, 1) <- .
95          BiAssocLow(DerivedStatistic, DerivedStatisticStrategy,
                DerivedStatisticStrategy, 1) <- .
96          BiAssocHigh(DerivedStatistic, DerivedStatisticStrategy,
                DerivedStatisticStrategy, 1) <- .
97
98          BiAssoc(Statistic, WeaponType) <- .
99          BiAssocHigh(Statistic, WeaponType, Statistic, 1) <- .
100
101         BiAssoc(Item, Inventory) <- .
102         BiAssocHigh(Item, Inventory, Item, 5) <- .
103         BiAssocHigh(Item, Inventory, Inventory, 1) <- .
104
105         BiAssoc(Item, CharacterClass) <- .
106
107         BiAssoc(Character, CharacterClass) <- .
108         BiAssocLow(Character, CharacterClass, CharacterClass, 1) <- .
109         BiAssocHigh(Character, CharacterClass, CharacterClass, 1) <- .
110
111         BiAssoc(CharacterClass, CharacterClass) <- .
112
113         BiAssoc(Character, SupportAffinity) <- .
114         BiAssocLow(Character, SupportAffinity, Character, 1) <- .
115         BiAssocHigh(Character, SupportAffinity, Character, 1) <- .
116         BiAssocLow(DerivedStatistic, SupportAffinity, SupportAffinity, 1) <- .
117         BiAssocHigh(DerivedStatistic, SupportAffinity, SupportAffinity, 1) <-
                .
118
119         ! x [ClassObject] y [ClassObject] : BiAssoc(y, x) <- BiAssoc(x, y).
120         ! x [ClassObject] y [ClassObject] z [nat] : BiAssocHigh(y, x, x, z) <-
                BiAssocHigh(x, y, x, z).
```

```
121         ! x [ClassObject] y [ClassObject] z [nat] : BiAssocHigh(y, x, y, z) <-
                BiAssocHigh(x, y, y, z).
122         ! x [ClassObject] y [ClassObject] z [nat] : BiAssocLow(y, x, x, z) <-
                BiAssocLow(x, y, x, z).
123         ! x [ClassObject] y [ClassObject] z [nat] : BiAssocLow(y, x, y, z) <-
                BiAssocLow(x, y, y, z).
124     }
125
126
127         //! x [ClassObject] y [ClassObject] : BiAssocDef(x, y) <=> BiAssocDef(
                y, x).
128 }
129
130 theory U:V {
131     ~ (? z [ClassObject] : BiAssoc(Loose, z)).
132 }
133
134 structure thestruct:V {
135 }
136
137 procedure main() {
138     print(modelexpand(T,thestruct)[1])
139 }
```

chap–rol–idp/defs.idp

Fiche masterproef

Student: Thomas Vochten

Titel: Automatische verificatie en kwaliteitscontrole van UML-diagrammen met FO(.)

Engelse titel: TBD

UDC: TBD

Korte inhoud:

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdoptie Gedistribueerde systemen

Promotor: Prof. dr. Marc Denecker

Assessor: TBD

Begeleider: Matthias van der Hallen