Софийски Университет „Св. Климент Охридски”
Факултет по математика и информатика
Специалност „Информатика”
Специализация „Логика и Алгоритми”

Дипломна работа

на

Димитър Тодоров Георгиев
специалност “Информатика”, ф. № М-21323

Тема:

Програмна реализация на алгоритъма SQEMA за модална определимост

Научен ръководител: проф. Димитър Вакарелов
доц. д-р Тинко Тинчев

София, 2006

Съдържание

1. Увод

В езика на едномодалната логика имаме изброимо много модални променливи (т.е. формулите са съставени от), булеви оператори (отрицание, конюнкция, дизюнкция) и модални оператори (необходимо и възможно). Един от възможните начини да се представи семантиката на едномодалните формули е чрез структури на Крипке. За да оценим дадена модална формула, представяме структура на Крипке <W,R>, където W представлява множество от светове, а R е бинарна релация между елементите на W, както и оценка на променливите v:VARxW -> {true, false}, където VAR е изброимото множество на променливите в езика. С други думи, оценката казва дали дадена променлива е вярна или не в даден свят. Семантично ще разбираме W като множество от светове, а релацията R като някакъв преход между световете. Дадена формула е необходимо вярна в един свят, т.с.т.к. тя е вярна при тази оценка във всички светове, до които може да се достигне от дадения свят чрез релацията R. Дадена формула е възможно вярна в един свят, т.с.т.к. тя е вярна при тази оценка в поне един от световете, до които може да се достигне от дадения свят чрез релацията R. В останалите (булевите) случаи дефиницията за вярност на формула е аналогична на дефиницията в класическата логика. Казваме, че една формула е вярна в дадена структура на Крипке, ако тя е вярна в тази структура на Крипке при произволна оценка на променливите.

Фундаментален въпрос в модалната логика е: При дадена модална формула да се намери формула от първи ред (с ЕР и равенство), която е вярна точно в онези структури на Крипке, в които е вярна дадената модална формула. Оказва се обаче, че

този въпрос е алгоритмично неразрешим. Съществуват различни опити за частичното решаване на въпроса, от решаване на частни случаи „на ръка", до сложни алгоритми като SCAN и DLS. Трябва да отбележим, че алгоритъмът SCAN е пълен за Sahlquist формули [3], [4], [5].

Новият алгоритъм SQEMA ([1], [2]), разработен от Димитър Вакарелов, Валентин Горанко и Willem Conradie, се справя с най-широка гама от формули. Той се справя по-добре от досегашните алгоритми. Той притежава и относителна леснота на прилагане. На примери с малък брой променливи (3-4) програмната реализация работи с много висока скорост.

Алгоритъмът работи по следния начин.

Въвежда се нов език, в който освен променливи съществуват и номинали. Номинал е специална променлива, която при всяка дадена оценка е вярна точно в една точка на `W`. Променяме малко семантиката на формулите. Оценката v се превръща във функция от `VAR` към `P(W)`, където `P(W)` е множеството от всички подмножества на `W`. Сега от структура на Крипке и оценка можем да съпоставим на всяка формула дадено подмножество на `W`, по правила, описани в теоретичната част. Една формула е вярна (в тази структура при тази оценка), ако е съпоставена на цялото множество `W`, което означаваме още с `1`. Празното множество означаваме още с `0`. Така например на всеки номинал съответства синглетон, който е подмножество на `W`. Така получаваме теоритико-множествено представяне на формулата (при фиксирана структура на Крипке и оценка на променливите).

В така променения език, разглеждаме отрицанието на формулата. Тя не е вярна тогава и само тогава, когато съществува свят от `W`, който не участва в нейното теоретико-множествено представяне. Следователно, съществува номинал C_1, при който формулата `A` не е вярна:

C_1 -> ¬A

След това се прилагат описаните в [1] и [2] правила и накрая се прилага адекватният вариант на Лемата на Акерман за елиминация на една от променливите, ако това е възможно, след което се продължава с друга променлива. Алгоритъмът въвежда и други номинали в частите на формулата по време на своята работа. Крайният резултат е формула без променливи (но с номинали). Понякога е невъзможно достигането на такъв резултат (тогава се казва, че алгоритъмът SQEMA се проваля на дадената формула). Алгоритъмът предвижда търсене с връщане (backtracking) като състоянията, които се запазват, са следните: редът, в който се елиминират променливите, моменти за вземане на дуалната на дадена променлива и редът, в който се изследват членовете на всяка дизюнкция. Backtracking се извършва при провал на алгоритъма. Крайният резултат, ако достигнем до такъв, се отрича отново. След това се превежда до формула от първи ред на три стъпки. Първо се получава формула, съдържаща множества, после множествата се елиминират по естествен начин, след което се елиминират някои равенства.

В тази дипломна работа се представя реализация на алгоритъма SQEMA, написана изцяло на езика Java. Реализирана е основната версия на алгоритъма. Като "екстра" се позволява използването на номинали на входа. Възможно е да се приложат доста подобрения, за да се обхванат повече формули – например да се напише алгоритъм за откриване на модални тавтологии или тъждествено неверни подформули (в момента се използва само доста сложна евристика на всяка стъпка от прилагането на правилата), както и проверка за монотонност на дадена подформула (засега се проверяват само случаите с променлива и отречена променлива). Също така е възможно да се направят

бъдещи версии, които позволяват работа с полиадични модалности (където релацията е на повече от два аргумента).

В реализацията съм се старал да използвам само библиотеките, които присъстват в JDK 1.1.8, въпреки че новите версии на Java, ако бъдат използвани с програмата, довеждат до значително подобряване на скоростта, заради по-добрите оптимизиращи JIT компилатори. Програмата работи по два различни начина: като Swing приложение (забележка Swing не е достъпен в стандартна дистрибуция на Java 1.1.8) и като WAR Web Application. Swing версията на програмата е тествана под Java 1.3.x и Java 1.4.2_09, а WAR версията работи нормално под Java 1.4.2_09 с Tomcat 4.1.31. Програмите работят и под Ubuntu Linux 5.10 (Breezy Badger) със Sun JDK 1.4.2_11 и Tomcat 3.3.2. Web Application-ът е достъпен от адрес

http://www.fmi.uni-sofia.bg/fmi/logic/sqema/

Освен това, Web Application-ът е сложен на безплатен хостинг в интернет на адрес:

http://myjavaserver.com/~dimitertg/sqema/

Присъстват още и компилиращ bat файл за Microsoft Windows® NT/2000/XP, който, при наличине на JDK 1.3.x или по-висока версия (би трябвало да работи и с JDK 1.2.x и даже JDK 1.1.8 с инсталиран Swing, но не е тествано) някъде на файловата система (не се изисква инсталиране, само наличието на файловете на JDK е достатъчно) и наличието на 'bin' директорията на това JDK в системната променлива PATH, компилира приложението и изготвя release архиви. При нужда този bat файл може лесно да се пренапише до bash script за някоя дистрибуция на GNU/Linux или до произволен скрипт за друга система.

Изходният код и работеща версия на програмата могат да се изтеглят от адрес:

http://myjavaserver.com/~dimitertg/

След което от лявото меню се избира „SQEMA" и после се сваля файлът, линк към който се появява от дясната страна на екрана.

Използвам възможността да благодаря на дипломните си ръководители, професор Димитър Вакарелов и доцент доктор Тинко Тинчев.

2. Теоретична част

2.1. ***Проблемна област, дефиниции и терминология***

Разглеждаме едномодални логики ([1] и [2]). Езикът се състои от: символи □ и ◊ , които представят модалностите необходимост и възможност; изброимо множество *VAR* от символи, които представят променливи; булеви логически знаци {¬(отрицание – в програмата се използва ~), &(конюнкция), |(дизюнкция), ->(импликация)} и скоби - (,) като помощни символи. Еквивалентността се разглежда като съкращение за конюнкцията на двете естествени импликации. Също така за по-добра четимост ще използваме символа **.** като разделител между модални оператори и техните аргументи, когато това е необходимо.

Модален език с номинали наричаме модален език, към който е добавено изброимо множество от номинали. Към дефиницията на формула добавяме една нова подточка – всеки номинал е формула.

За целите на работата на алгоритъма добавяме към езика оператори $\Box^{-1}$ и $\Diamond^{-1}$, които са съответно обърната необходимост и обърната възможност.

Деф. ***Формула*** е крайна редица от символи, която се получава със следната индуктивна дефиниция:

- Променливи са формули;
- Ако A е формула, то ¬A е формула;
- Ако A и B са формули, то (A σ B), където σ ∈ {**&, |, =>**}, е формула;

- Ако A е формула, то μA, където $\mu \in \{\Box, \Diamond, \Box^{-1}, \Diamond^{-1}\}$, е формула.

Деф. ***Крипке структура*** е наредена двойка <*W, R*>, където *W* е непразно множество, а *R* е бинарна релация в *W*. *W* наричаме ***универсум*** или ***множество от възможните светове,*** а бинарната релация *R* е релацията на достижимост.

Деф. ***Оценка*** в една Крипке структура е функция *v*, която на всяка променлива съпоставя множество от възможни светове (т.е. подмножество на *W*). Съдържателно: *v*(*p*) е множеството от всички светове, в които *p* е вярна.

Деф. ***Модел*** е двойка от Крипке структура и оценка в нея.

Деф. ***Вярност*** на формула A в свят *w* от модел дефинираме с индукция относно построението на формулите:

- Нека A е променлива *p*. A е вярна точно в онези светове, които са от *v*(*p*);
- Нека A е ¬B. A е вярна точно в онези светове от *W*, в които B не е вярна;
- Нека A е B & C. A е вярна точно в онези светове, в които са едновременно верни B и C;
- Нека A е B | C. A е вярна точно в онези светове, в които е вярна поне една от B и C;
- Нека A е B -> C. A е вярна точно в онези светове от *W*, в които не е вярна B или е вярна C;
- Нека A е □B. A е вярна точно в онези светове, които имат свойството в достижимите от тях с *R* светове да е вярна B;

- Нека A е ◊B. A е вярна точно в онези светове, които имат свойството да съществува достижим от тях с R свят, в който да е вярна B;
- Нека A е $\Box^{-1}$B. A е вярна точно в онези светове, които имат свойството в достижимите от тях с обратната релация на R светове да е вярна B;
- Нека A е $\Diamond^{-1}$B. A е вярна точно в онези светове, които имат свойството да съществува достижим от тях с обратната релация на R свят, в който да е вярна B;

Казано по друг начин, разширяваме оценката v до функция с дефиниционна област множеството на всички формули и верността на формула A в свят w е верността на $w \in v$(A). Това разширение е следното:

- $v(\neg B) = W \setminus v(B)$
- $v(B \ \& \ C) = v(B) \cap v(C)$
- $v(B \mid C) = v(B) \cup v(C)$
- $v(B \text{ -> } C) = (W \setminus v(B)) \cup v(C)$
- $v(\Box B) = \{ x \in W \mid \forall y(xRy \Rightarrow y \in v(B)) \}$
- $v(\Diamond B) = \{ x \in W \mid \exists y(xRy \ \& \ y \in v(B)) \}$
- $v(\Box^{-1} B) = \{ x \in W \mid \forall y(yRx \Rightarrow y \in v(B)) \}$
- $v(\Diamond^{-1} B) = \{ x \in W \mid \exists y(yRx \ \& \ y \in v(B)) \}$

Да отбележим следното твърдение, на чието доказателство няма да се спираме, поради това, че се състои в непосредствена проверка.

Тв. Нека v и v' са оценки в една Крипке структура. Нека A е формула и $v(p) = v'(p)$, за всяка променлива p която участва в A. Тогава $v(A) = v'(A)$

Поради тази причина в моделите, които използваме, няма да отбелязваме стойностите на променливите, които не се срещат в

разглежданата формула. Освен това няма да задаваме стойностите на променливите, когато е ясно, че те не влияят на верността на формулата, макар да се срещат в нея. Например, ако формулата е A | B и сме задали стойности на променливите в A, при които тя е вярна, няма да задаваме стойностите на променливите, участващи в B.

Деф. Една формула A се нарича ***вярна в даден модел***, ако е вярна във всеки негов свят и се нарича ***изпълнима в модела***, ако е вярна поне в един свят.

Деф. Една формула A се нарича ***вярна в Крипке структура***, ако е вярна във всеки модел над тази структура и се нарича ***изпълнима в структурата***, ако е вярна поне в един модел над нея.

2.2. Описание на алгоритъма SQEMA

2.2.1. **Лема на Акерман (модална форма) [1] и [2]**

Нека A, B(p) са формули в разширения език на модалната логика, съдържащ номинали и обърнати модалности, и нека променливата p не се съдържа в A, а B(p) е негативна в p. Тогава за всеки модел M:

$M \models B(A) \leftrightarrow M' \models (A \rightarrow p) \ \& \ B(p)$

За някой модел M', който е различен от M най-много в оценката на p.

Доказателство:

Ако $M \models B(A)$, тогава $M' \models (A \rightarrow p) \ \& \ B(p)$ за модел M' такъв, че оценката на p в M е равна на оценката на A в M.

Обратно, ако M' ⊨ (A → p) & B(p) за някой модел M', то M ⊨ B(A/p), защото B(p) е монотонна надолу. Оттук M ⊨ B(A/p).

Трябва да отбележим, че в програмната реализация се използва видоизменена лема на Акерман:

M ⊨ B(A) ↔ M' ⊨ (A | p) & B(¬p)

Което е различен запис на доказаното в [1] и [2].

2.2.2. **Описание на алгоритъма SQEMA, според [1] и [2]**

1. Вход: модална формула φ

2. Формулата φ се отрича и ¬φ се преобразува в "негационна нормална форма", елиминирайки импликациите и еквивалентностите и преобразувайки формулата така, че всички знаци за отрицание да са само пред променливи.

3. Разпределят се възможностите и конюнкциите над дизюнкции колкото е възможно, използвайки еквиваленциите

◊(φ | ψ) ≡ (◊φ | ◊ψ)

(φ | ψ) & θ ≡ (φ & θ) | (ψ & θ)

4. Избира се начален номинал i (който не присъства в нито една подформула) и за всеки дизюнкт 'a' прилагаме независимо останалата част от алгоритъма, както следва.

5. Записва се i -> a.

6. Елиминира се всяка променлива, относно която системата е позитивна или негативна, като се заменя съответно с 1 или 0.

7. Ако са останали променливи в системата, избира се една за елиминиране – 'p', елиминирането на която още не е било пробвано, и се продължава към стъпка 8. Изборът се извършва недетерминирано. Ако всички останали променливи са били пробвани и стъпка 8 е пропаднала, докладва се провал. Ако всички променливи са били елиминирани от системата, се продължава към стъпка 9.

8. Целта тук е, прилагайки трансформиращите правила, описани по-долу, да се пренапише системата от уравнения така, че правилото на Акерман да стане приложимо за избраната променлива p, за да може тя да бъде елиминирана. Така целта е да се трансформира системата в такава, в която всяко уравнение е или негативно относно p, или под форма на a -> p, където p не се съдържа в ‘a’, т.е. да се ‘извлече’ p и да се ‘реши’ системата за p. Ако това е невъзможно, прави се backtracking, сменя се поляритета на p като се замества с ¬p навсякъде и се опитва отново подготовка за прилагане на правилото на Акерман. Ако този опит също пропадне, се осъществява връщане до стъпка 7.

9. Ако тази стъпка е била достигната от всички клонове, започнати в стъпка 4, значи всички променливи са били елиминирани. Остава резултатът да се запише като формула от първи ред. В рамките на всяка от системите, започнати на стъпка 4, формулите се обединяват в конюнкция ‘pure’ и се формира формула

∀(y1, …, yn) ∃(x0) (ST(¬pure)),

където y1, …, yn са променливи от първи ред, съответстващи на номинали, но yi, което съответства на началния номинал i се оставя несвързана – защото при търсене на локално условие кванторът не е за всеобщност. x0 се записва като равно на yi. Взима се конюнкцията на тези преводи на системи през всички дизюнктивни клонове.

2.2.3. **Променен, но еквивалентен, алгоритъм SQEMA за целите на реализацията**

Промените са предложени от Димитър Вакарелов и Тинко Тинчев.

1. Стъпка 1 остава същата.

2. Стъпка 2 се извършва преди отрицанието.

3. Стъпка 3 също се извършва преди отрицанието. Разпределят се необходимостите над конюнкции, доколкото е възможно, използвайки еквиваленцията:

$\Box(\varphi \,\&\, \psi) \equiv (\Box\varphi \,\&\, \Box\psi)$.

След това се образува КНФ според тази еквиваленция:

$(\varphi \,\&\, \psi) \mid \theta \equiv (\varphi \mid \theta) \,\&\, (\psi \mid \theta)$.

Трябва да отбележим, че тук, за разлика от стъпки 5 и 7|8, се използва различен алгоритъм за КНФ, който не извлича ромбчета пред дизюнкции.

За да се избегнат усложнения от разделената обработка, стъпки 2 и 3 се повтарят, докато се достигне формула, еднаква с предходната. Разликата в скоростта на изпълнение на проведените тестове, породена от повторенията, е незначителна.

4. Стъпка 4 отново се извършва преди отрицанието. Избира се начален номинал C_1 и за всеки конюнкт A поотделно се изпълнява останалата част от алгоритъма.

5. Тук вече извършваме отрицанието. Записваме

$\neg C_1 \mid \neg A$.

Записваме в КНФ в дълбочина (т.е. през модалности) като се стремим да извлечем възможностите пред дизюнкциите, с цел да приложим по-лесно правилото за елиминиране на възможност и образуване на релационна двойка с нов номинал.

6. Тази стъпка остава същата.

7 и 8. Тези стъпки остават почти същите, само правилата са модифицирани да работят с дизюнкция, вместо с импликация. Също така, при всяко успешно елиминиране, правим т.нар. “нормализация” на формулата, която включва опит за опростяване, КНФ (с извличане на ромбчета) и елиминиране на променливи. За по-подробна информация, вижте описанието на реализацията – метод usePermutation на клас SQEMA.

9. Резултатите от всяка система се обединяват в конюнкция, тя се отрича и преминава през normalizeResult метода на SQEMA - нормализация на резултата, която се опитва да елиминира последната променлива, ако тя е участвала само в alpha или beta правило. Тази нормализация е подобна на предишната, но не се преминава през КНФ с цел да не се усложнява резултатът. След това, резултатите се обединяват в конюнкция и се превеждат до формула от първи ред. Избягва се използването на x0, а вместо това се ползва директно несвързаната променлива x.

2.2.4. **Оригиналните правила за преобразуване в SQEMA, [1] и [2].**

2.2.4.1. AND-правило

A -> B & C → A -> B, A -> C

2.2.4.2. OR-правила

A -> B | C → (A | ¬B) -> C

(A | ¬B) -> C → A -> B | C

2.2.4.2. □-правила

A -> □B → $\Diamond^{-1}$A -> B

$\Diamond^{-1}$A -> B → A -> □B

2.2.4.3. ◊-правило

j -> ◊B → j -> ◊k, k -> B,

където j е някой номинал, а k е нов номинал.

2.2.5. **Оригиналното правило на Акерман, [1] и [2].**

A1 -> p, …, An -> p,

B1(p), …, Bm(p)

→

B1[(A1 | … | An) / p], …, Bm[(A1 | … | An) / p],

където p не участва в Ai, Bj са негативни в p, няма други участия на p в системата.

2.2.6. **Модифицираните правила за преобразуване в SQEMA.**

2.2.4.1. AND-правило

A | (B & C) → A | B, A | C

2.2.4.2. OR-правило (обърнатото не се използва от реализацията, тъй като реализацията записва backtracking точка при срещане на дизюнкция с участието на променливата в двата клона, вж. описанието на SQEMA.usePermutation).

A | (B | C) → (A | B) | C

2.2.4.2. □-правила

A | □B → $□^{-1}$A | B

A | $□^{-1}$B → □A | B

2.2.4.3. ◊-правило

¬c | ◊B → c -> ◊d, ¬d | B,

където c е някой номинал, а d е нов номинал.

2.2.7. **Модифицираното правило на Акерман.**

A1 | p, …, An | p,

B1(¬p), …, Bm(¬p)

→

B1[(A1 & … & An) / ¬p], …, Bm[(A1 & … & An) / ¬p],

където p не участва в Ai, Bj са негативни в p, няма други участия на p в системата.

2.2.8. **Коректност на модифицираните правила**

Твърдение: Модифицираните правила са коректни, т.е. преобразуванията са локално еквивалентни за даден модел.

Доказателство: Очевидно е за AND и OR правилата.

Разглеждаме първото □-правило, за другото доказателството е аналогично.

M ⊨ A | □B. Ще докажем, че M ⊨ $\square^{-1}$A | B. Обратното е еквивалентно на второто □-правило и доказателството е аналогично.

Нека c е точка от света на модела M. Ще докажем, че M,c ⊨ $\square^{-1}$A | B, откъдето следва твърдението.

Ако съществува y: yRc, то от M,y ⊨ A | □B следва M,c ⊨ $\square^{-1}$A | B. Ако не съществува y: yRc, то M,c ⊨ $\square^{-1}$A е тривиално вярно.

Разглеждаме ◊-правилото.

M ⊨ ¬c | ◊B. Ще докажем, че M' ⊨ c -> ◊d, ¬d | B, където M' е нов модел, който се различава от M най-много в оценката за новия номинал d.

$\exists y(cRy \,\&\, y \in B)$

Нека с d означим нов номинал и променим оценката в M до M', където $v(d) = y$. Така получаваме търсеното

M' ⊨ c -> ◊d, ¬d | B.

Обратно, нека M' ⊨ c -> ◊d, ¬d | B, където M' е нов модел, който се различава от M най-много в оценката за новия номинал d. Тогава

cRd, M', d ⊨ B.

Следователно

$\exists y(cRy \,\&\, y \in B)$

т.е. M' ⊨ c -> ◊B, сл. M ⊨ c -> ◊B, тъй като d не се среща в (c -> ◊B).

3. Описание на програмната реализация

За реализиране на програмата е използвана безплатната версия на развойната среда на **ProSyst** "***mBedded Builder"*** (вече недостъпна от интернет) и по-конкретно езикът Java. Избрах тази развойна среда, заради относителната простота на работата с нея, както и заради моето участие в нейното написване.

Програмата работи на 3 етапа:

1. Конвертиране на данните от символен низ до обект от тип FormulaPart, представящ модална формула.
2. Прилагане на алгоритъма SQEMA върху така представената модална формула.
3. Конвертиране на резултата от работата на SQEMA до формула от първи ред. Използват се някои евристики за опростяване на резултата.

3.1. Класове, реализиращи синтактичния анализ и представянето.

Входните данни на програмата представляват въведен от потребителя символен низ, представящ дадена формула.

3.1.1. Лексически и синтактичен анализ

org.modallogic.parser.CupLexer

Извършва лексически анализ на формулите, разделяйки ги на символи, които се използват от CupParser.cup

public void init(String text)

Извършва лексическия анализ чрез едно прочитане на параметъра 'text'. Резултатите се запазват във Vector за по-късно итериране.

public Symbol next_token() throws Exception

По спецификацията на java_cup.runtime.Scanner. Итерира по символите, получени от init, връщайки ги един по един на CupParser.cup.

org.modallogic.parser.CupParser

Този клас съществува като файл ‘CupParser.cup’. Самият клас се генерира по време на компилацията от Java CUP. CupParser.cup е описание на контекстно-свободна граматика на модални формули в BNF-подобна нотация. Граматиката съдържа терминалите лява и дясна скоба, отрицание (~), възможност (@), необходимост (#), конюнкция (&), дизюнкция (|), изключващо ‘или’ (+), импликация (->), обърната импликация (<-), еквиваленция (<->), дума–променлива (дума, състояща се от малки латински букви и цифри и започваща с малка латинска буква, различна от ‘c’), дума-номинал (състояща се от малки латински букви и цифри и започваща с малка латинска буква ‘c’), единица (1), нула (0). Нетерминалите са: модална формула (наричана по-късно за по-кратко ‘формула’), формула в скоби, отречена формула, възможна формула, необходима формула, конюнкция на две формули, дизюнкция на две формули, изключващо ‘или’ на две формули, импликация на две формули, обърната импликация на две формули, еквиваленция на две формули, променлива, номинал, единица, нула. Аксиомата на граматиката е модална формула. За лексически анализ се използва org.modallogic.parser.CupLexer. Обектите, генерирани от CupParser, са от класове, които са наследници на org.modallogic.formula.FormulaPart. За повече информация, вж. описанието на пакетите org.modallogic.formula и org.modallogic.firstorder.formula.

org.modallogic.parser.Lexer

org.modallogic.parser.Parser

Тези класове съществуват за улеснение на компилацията на програмата от някои среди. Тъй като org.modallogic.parser.CupLexer не може да бъде компилиран без CupSymbols, който се получава от CupParser.cup по време на пълно компилиране (което използва org.mocallogic.parser.build.MakeParser), а самият CupParser изобщо не съществува по време на разработка, се използва динамично зареждане на CupLexer и CupParser, за да се избегне компилационната зависимост на цялата програма от CupLexer и CupParser.

org.modallogic.parser.build.MakeParser

Използва се от build.bat за генериране на CupParser. Компилирането протича както следва: първо се компилира MakeParser, после се изпълнява, за да генерира CupParser, след това се компилира останалата част от програмата.

org.modallogic.parser.test. ParserTest

Програма, останала от първите стадии на разработка. Използва CupParser за синтактичен анализ на примерна формула и на резултата се прилагат някои прости алгоритми от org.modallogic.algorithm.

3.1.2. **Представяне на формулите чрез обекти**

org.modallogic.formula.FormulaPart

Основният клас за работа с формули – както модални, така и от първи ред. Всяка формула е Tuple, на първо място стои идентификация на типа (Integer), след това са т.нар. наследници (children) – обекти от тип FormulaPart. Смисловото интерпретиране на формулите не е реализирано с полиморфизъм. За да не се използва Java операторът instanceof, който често е синтактично неудобен, има метод is, на който се подава обект от тип Class. Това елиминира възможността класовете за формули

да се подредят в класова йерархия, но е напълно достатъчно за целите на програмата.

Съществуват две групи от наследници на FormulaPart. В едната са класовете от пакет org.modallogic.formula, а в другата – класовете от пакет org.modallogic.firstorder.formula.

Формулите поддържат сортиране на наследниците си, което позволява бързо да се откриват лексически еднакви с точност до пренареждане формули. Разбира се, еднакви формули от вида

((A & B) & C) и (A & (B & C))

ще бъдат счетени за различни при този подход, но за откриване на такива ситуации са предвидени класовете DisjunctionMultiple (единственият клас от тази йерархия, който поддържа операции) и ConjunctionMultiple. В програмата се използват евристики за елиминиране на еднакви формули, които включват получаване на конюнктивна нормална форма – чрез използването на операциите в DisjunctionMultiple се получава опростяване на формулите.

Въведената наредба на формулите по тип също е от значение – трябва да се обърне внимание на класовете Constants – по един в двата пакета с формули. Тази наредба се използва съществено от основния алгоритъм SQEMA, който разчита на това, че отречените номинали стоят на първо място в дадена формула. За целите на алгоритъма е въведен и клас DisjunctionOrdered, обектите от който не сортират двата си наследника. Този клас се използва от основния алгоритъм, който записва в такива обекти обърнатите дизюнкции за backtracking.

3.2. Класове, реализиращи алгоритмите

В реализацията има три основни групи алгоритми:

1. Алгоритми за опростяване и преобразуване на модални формули.

2. Основният алгоритъм SQEMA.

3. Алгоритми за преобразуване на резултата до формула от първи ред и опростяване на така получения превод.

3.2.1. **Входната точка на алгоритъма**

org.modallogic.conversion.Conversion

Входната точка на алгоритъма е осъществена в този клас. Той се грижи за правилната последователност на извикване на алгоритмите, получавайки от вход два вида изход – междинен резултат с номинали и краен резултат – формула от първи ред.

Ако алгоритъмът пропадне, се генерира изключение.

3.2.2. **Опростяване и преобразуване на модални формули**

org.modallogic.algorithm.Cleaner

Този алгоритъм опростява формулите по следната логика: премахват се всички операции освен конюнкция, дизюнкция, отрицание, необходимост, възможност. Отрицанията се изместват възможно най-навътре. Обработват се частните случаи с единица и нула.

org.modallogic.algorithm.CNFToFormula

Премахване на DisjunctionMultiple и ConjunctionMultiple от дадена формула. DisjunctionMultiple и ConjunctionMultiple се получават при намиране на конюнктивна нормална форма.

org.modallogic.algorithm.FlatConverter

Този алгоритъм извършва обратната операция на предходно описания. Тук се стремим да получим „плоска” формула, подходяща за показване на потребителя. За разлика обаче от конюнктивната нормална форма, тази „плоска” формула не е

излишно усложнена. Използва се за форматиране на изход. Съдържа известно количество опростяване на формулата.

org.modallogic.algorithm.PartialCNF

Този алгоритъм получава конюнктивна нормална формула – тъй като имаме модални оператори, алгоритъмът работи и в дълбочина, т.е. след всеки модален оператор стои формула в КНФ. Съществено се използват опростяващите възможности, вградени в класа DisjunctionMultiple.

3.2.3. **Основният алгоритъм SQEMA**

org.modallogic.algorithm.SQEMA

Този клас използва статични методи, за да реализира самия алгоритъм SQEMA.

public static FormulaPart sqema(FormulaPart modalFormula)

Входната точна на алгоритъма. Аргументът е модална формула, която трябва да се преобразува по алгоритъма SQEMA до модална формула без номинали. Ако алгоритъмът не може да намери решение, резултатът от изпълнението е null. Формулата се опростява, превръща се в конюнктивна нормална форма, разпределят се квадратчета над конюнкции, избира се стартов номинал от нов номинален контекст, всяка елементарна дизюнкция (т.е. всеки конюнкт) се преобразува от метода partialSQEMA, резултатите се обединяват в конюнкция и получената ConjunctionMultiple се връща като резултат от изпълнението на програмата. Ако partialSQEMA пропадне на някой от конюнктите, резултатът от sqema е null.

```
private static FormulaPart partialSqema(
        FormulaPart modalFormula,
        NominalContext nominalContext,
        Nominal startingNominal)
```

Този метод работи по следния начин: Първо формулата се отрича и нормализира (с метода normalize). След това се намират променливите, които предстоят да бъдат елиминирани (с метода findToEliminate). Ако променливи отсъстват, се връща като резултат отрицанието на дизюнкцията на отрицанието на стартовия номинал и формулата. Иначе, променливите се сортират, създава се обект от клас SQEMA, който имплементира PermutationUser и се стартира алгоритъмът за намиране на пермутации (от пакет org.modallogic.util.permutation). Ако бъде намерена пермутация на променливите, която елиминира всички променливи, резултатът се връща. Иначе, резултатът е null.

public boolean usePermutation(int[] indices)

Това е основният метод, който реализира алгоритъма. Параметърът е масив от индекси на променливи, които трябва да бъдат елиминирани в този ред. Ако методът пропадне, резултатът е false, което казва на генератора на пермутации да продължи със следващата пермутация. Ако се получи резултат, резултатът от метода е true. Методът е специфициран от интерфейса PermutationUser.

Методът запомня междинни данни от своята работа, за да може да осъществи търсене с връщане в рамките на дадената пермутация от променливи. Използва се вътрешният клас BacktrackContext, който съдържа: обработените подформули, оставащите подформули, текущата дизюнкция (в случай, че се запомня обърната дизюнкция), номинален контекст (копие на оригинала, за да може да се премахнат създадените от друг клон номинали), индекс в пермутацията, указващ индиректно текущата променлива, флаг за обръщане на поляритета – когато се записва контекст за обръщане на поляритета на дадена променлива.

Работата протича по следния начин: Създава се стек от BacktrackContext обекти, създават се две празни множества – от обработени и необработени подформули. В множеството на необработените формули се добавя дизюнкцията на отречения стартов номинал и началната подформула. Помни се и контекстът, до който трябва да се върнем (в началото такъв няма). Започва цикъл по променливите за елиминиране. На пръв поглед този цикъл върви само напред, но заради осъществяването на backtracking може да се случи да се върнем на предишна променлива. В цикъла избираме променлива според индекса в пермутацията и запазваме един backtracking контекст – за обръщане на поляритета. Започва вътрешен цикъл, без условие. В него първо гледаме дали трябва да възстановим контекст. След това, докато има останали формули за обработка с текущата променлива, избираме една от формулите за обработка. Ако в нея не се среща търсената променлива, я обявяваме за обработена. Ако формулата отговаря на алфа или бета условието за прилагане на лемата на Акерман, обявяваме я за обработена.

Ако формулата е дизюнкция и променливата не се среща в дясната страна на дизюнкцията, обръщаме дизюнкцията.

Ако формулата е дизюнкция и десният й наследник също е дизюнкция, гледаме дали търсената променлива има позитивно участие в лявата част на вътрешната дизюнкция. Ако има, но не участва позитивно в дясната част на вътрешната дизюнкция, обръщаме вътрешната дизюнкция и добавяме така получената подформула към списъка на необработените. Ако има, но участва позитивно и в дясната част, тогава се налага да запишем backtrack контекст за по-късна обработка, съдържащ обърната версия на вътрешната дизюнкция.

След това, прилагаме правилата към формулата чрез метода applyRules. Този метод ще реши дали формулата е обработена напълно спрямо тази променлива или има нужда от още обработка.

Щом обработим всяка подформула доколкото е възможно, проверяваме дали можем да приложим лемата на Акерман спрямо тази променлива. Ако поне една от подформулите не отговаря на условията (които са три: да не съдържа въпросната променлива или да отговаря на алфа или бета условията на Акерман лемата) – то алгоритъмът пропада за тази променлива и се прави опит за backtracking. Ако подформулите, съдържащи променливата, са само от тип алфа или от тип бета, Акерман субституция не се извършва, но следващата нормализация на системата отстранява променливата, заместваики я с константа. Ако няма възможност за backtracking в рамките на тази пермутация на променливите, обявяваме пермутацията за провал. Ако подформулите отговарят на условията, елиминираме текущата променлива и продължаваме със следващата според избраната пермутацията като преди това подготвяме подформулите чрез метода normalize, който прави опит за опростяване на подформулите.

Ако приключим успешно с алгоритъма SQEMA за избраната пермутация, обединяваме резултата в конюнкция, отричаме я и нормализираме резултата чрез normalizeResult, след което обявяваме текущата пермутация на променливите за успешна, за да може генераторът на пермутации да преустанови своята работа.

```
public static void applyRules(
        NominalContext nominalContext,
        FormulaPart toProcess,
        SetHash processed,
```

SetHash remaining

)

Прилага едно от правилата за извод на алгоритъма SQEMA спрямо дадената подформула. След прилагане на приложимото правило, методът слага получените подформули в processed или remaining множествата според това дали съответната получена подформула може да бъде обработена повече или не. Отречени или положителни променливи и номинали се обявяват за обработени. Релационна двойка на номинали се обявява за обработена. Дизюнкция, в която отдясно стои възможност, но отляво има нещо, различно от отречен номинал, се обявява за обработена. Ако подформулата не подлежи на обработка чрез правилата за замяна на алгоритъма SQEMA, тя се обявява за обработена.

Нормализация на формули.

По време на работа на SQEMA се използват следните видове нормализация на формули:

private static void normalize(SetHash system)

Нормализира множество от формули като първо ги обединява в ConjunctionMultiple, после нормализира тази конюнкция чрез метода normalize(FormulaPart) и резултатът, ако не е от тип (отречен номинал или формула), се представя като (0 или формула).

private static FormulaPart normalize(FormulaPart formula)

Нормализира единствена формула по следния алгоритъм: Формулата се изчиства с Cleaner, после се превръща в КНФ чрез PartialCNF, след това се премахват плоските конюнкции и дизюнкции чрез CNFToFormula, накрая се елиминират променливите, които могат да бъдат елиминирани непосредствено (чрез findToEliminate и removeVariables) – според статията, описваща алгоритъма SQEMA. Процесът се повтаря

изцяло, докато се получи формула, еднаква с предходната. След това така полученият резултат се връща.

private static FormulaPart normalizeResult(FormulaPart formula)

Използва се за нормализиране на резултат от работата на SQEMA. Формулата се изчиства с Cleaner, след това се прави опит за елиминация на променливи както в normalize(FormulaPart). Процесът се повтаря изцяло, докато се получи формула, еднаква с предходната. След това така полученият резултат се връща.

```
private static boolean lemmaAckermannAlpha(
  FormulaPart formula,
  Variable variableToEliminate,
  Not notVariableToEliminate
)
```

Проверява дали дадена формула изпълнява alpha-условието за лемата на Акерман, а именно: дали е от вида (A или p) или (p или A), където p е текущата променлива, а A не съдържа p.

```
private static boolean lemmaAckermannBeta(
  FormulaPart formula,
  Variable variableToEliminate,
  Not notVariableToEliminate
)
```

Проверява дали дадена формула изпълнява beta-условието за лемата на Акерман, а именно: дали е от вида B(not(p)). Това става като се проверява дали формулата съдържа not(p) и не съдържа неотречено p.

3.2.4. **Конвертиране на резултата от работата на SQEMA до формула от първи ред**

org.modallogic.firstorder.formula

Тази плоска йерархия съдържа класове, представящи формули от първи ред и символно представяне на множество.

org.modallogic.firstorder

Този пакет съдържа класове, които чрез статични методи извършват преобразуване на модална формула до формула от първи ред.

org.modallogic.firstorder.SetTranslator

public static FormulaPart convert(FormulaPart formula)

Този метод конвертира модална формула без променливи, но с номинали, до формула от първи ред. За всеки номинал се създава променлива от първи ред, началният номинал получава променлива x, а всеки следващ – променлива y1, y2, ... и т.н. Променливи с имена z1, z2, … и т.н се използват за означаване на временни променливи. Част от тези z-променливи изчезват при по-нататъшната обработка и затова се налага накрая да ги заменим с нови, наредени от 1 – това се извършва в клас org.modallogic.firstorder.TemporaryConverter. Формулата се преобразува до формула, описваща множество, и така полученото описание на множество се записва, че е различно от празното. След това отпред се записват квантори за всеобщност по поменливите y, а променливата x се оставя несвързана. Самото конвертиране до формула от първи ред протича на два етапа – в първия етап се опитваме да представим формулите от вида (A | B) като (conv(A) ≠ Ø | conv(B) ≠ Ø), с цел резултатът да съдържа по-малко множества. Това по-късно се оказа излишно усложнение, когато беше въведено елиминиране на множествата в записа на резултата, но е оставено в реализацията. Конвертирането се извършва с едно рекурсивно минаване по формулата.

org.modallogic.firstorder.SetEliminator

Този клас съдържа статичен метод, който конвертира формула от първи ред, съдържаща множества, до формула от първи ред без множества. Конвертирането се извършва с едно рекурсивно минаване по формулата.

org.modallogic.firstorder.EqualityEliminator

Този клас съдържа статичен метод, който елиминира някои равенства и неравенства от формула от първи ред, несъдържаща множества. Елиминират се следните равенства и неравенства: Елементарно верните/неверните ($x = x$, $x \neq x$), както и равествата, които участват в следния израз:

(Съществува y) ($x = y$ & phi(x, y))

Също така, тук се елиминират и квантори, чиито свързани променливи не участват във вътрешната им формула (т.е. били са елиминирани при някое опростяване).

Целият процес се повтаря, докато се получи формула, еднаква с предходната.

org.modallogic.firstorder.TemporaryConverter

Този клас съдържа статичен метод, който замества в една формула от първи ред променливите zi с нови променливи z1, z2, … и т.н., започвайки с нова z1 при първата срещната променлива zi и т.н. Използва се java.util.Hashtable за запомняне на съответствията между старите и новите означения. По този начин се преномерират всички z променливи за по-пригледно показване на потребителя. Конвертирането се извършва с едно рекурсивно минаване по формулата.

3.3. Реализация на потребителския интерфейс

Програмата има два различни вида потребителски интерфейси: SWING и Web/Servlet.

3.3.1. **Класове, реализиращи SWING потребителския интерфейс**

org.modallogic.gui.SQEMAGUI

Основният GUI клас, в него е реализиран основният потребителски интерфейс.

org.modallogic.gui.FormulaCouplePanel

org.modallogic.gui.FormulaPanel

Класове съответно за редактиране и изобразяване на модални формули. Първият клас използва втория за изобразяване на формулата, която се редактира.

org.modallogic.gui.Formulae

Помощен клас, който изчита файл "formulae.txt" от директорията, в която е стартирано приложението (в нормалния случай това е директорията "bin"), преобразува ги до flat формули (чрез FlatConverter) и представя резултата като масив от обекти от тип FormulaPart. Използва се за зареждане на стандартните стойности в приложението.

3.3.2. **Клас, реализиращ Web/Servlet потребителския интерфейс**

org.modallogic.servlet.SQEMAServlet

Осъществява целия потребителски интерфейс на Web приложението. Възможни са подобрения в стила му, както и прилагане на MVC (напр. чрез struts) в бъдещи версии.

3.4. Един интересен помощен клас

org.modallogic.util.sort.Tuple

Този клас е предназначен да представи неизменяема крайна наредена последователност от неизменяеми обекти, които са Comparable и могат да служат като ключ в Hashtable според

своята стойност (например, както е използвано тук, java.lang.String и java.lang.Integer). Тази последователност също е Comparable и може да се използва като ключ в Hashtable, т.е. има подходящо дефинирани hashCode и equals методи според съдържанието на обекта. Казваме, че два обекта са "сравними", ако и двата са Comparable и имат подходящо дефинирани методи hashCode, equals и compareTo, които успешно сравняват двата обекта, без да възниква Exception (напр. ClassCastException). Условието за конструиране на обекти от тип Tuple, които ще бъдат сортирани заедно, ще участват в една и съща колекция или ще бъдат ключове в една и съща Hashtable е следното: Първите (ако има такива, защото допускаме празни Tuple) обекти трябва да са сравними. Ако два съответни обекта на по-ниска позиция са равни, то съответните следващи трябва да са сравними. Например, в йерархията на FormulaPart, на първо място винаги стои Integer, указващ приоритета при сортиране на даден тип формула. По този начин се избягва сравняване на обекти от различни типове и възникване на ClassCastException. Тук не сме използвали java.util.TreeMap (клас, който отсъства в Java 1.1.8), но трябва да отбележим, че обектите от тип Tuple, ако са създадени с подходящи обекти, са добри ключове за TreeMap.

3.5. Други помощни класове

org.modallogic.util.sort.QuickSort

Тук има реализиация на оптимизирания алгоритъм за бърза сортировка, описан на C++ и C от Leendert Ammeraal в книгата "Algorithms and Data Structures in C++" [6]. Променена е логиката за сортировка, при подмасиви с по-малко от 50 елемента се преминава на специално оптимизирано сортиране с вмъкване, което използва двоично търсене. Някои тестове показват, че така модифицираният алгоритъм е по-бърз при големи масиви данни.

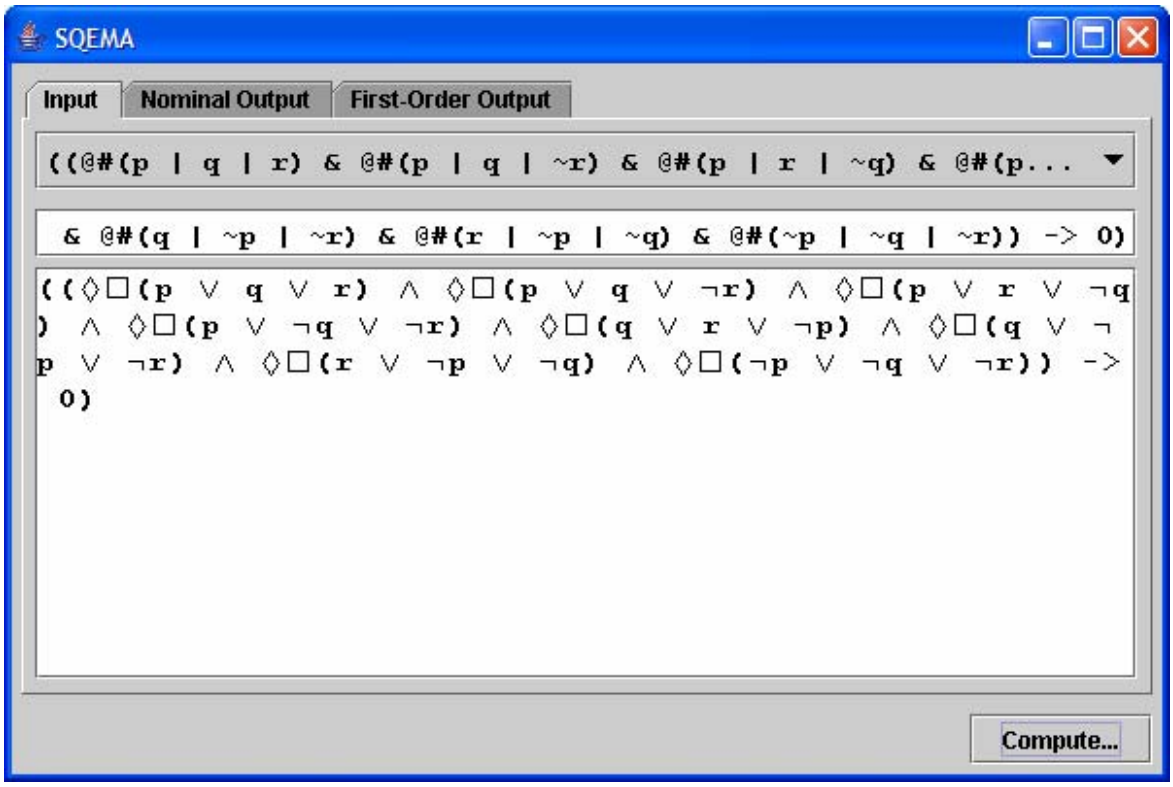
В реализацията на SQEMA е почти сигурно, че повечето сортировки използват основно сортирането с вмъкване.

org.modallogic.util.permutation. Permutations

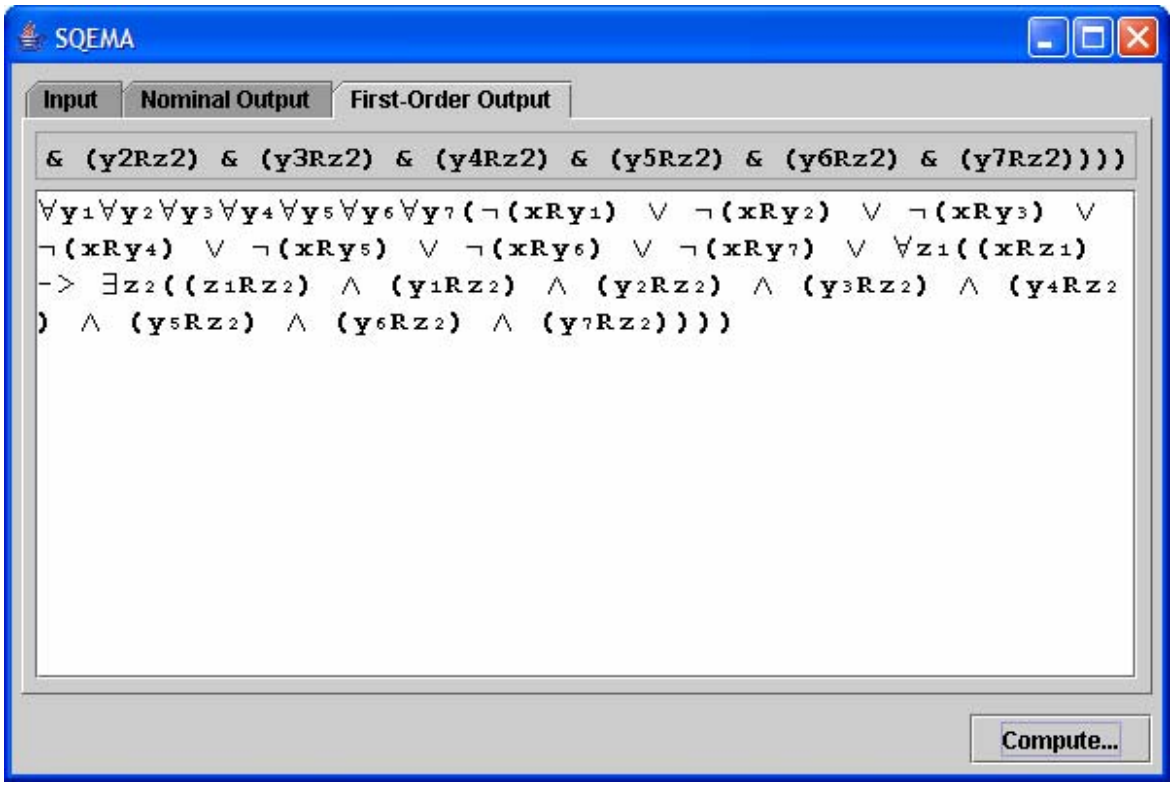
Тук има реализиация на алгоритъма за генериране на пермутации, описан от Преслав Наков и Панайот Добриков в книгата “Програмиране = ++Алгоритми;” [7]. Реализацията е променена, така че да е подходяща за търсене на подходяща пермутация сред всички възможни.

4. Ръководство за потребителя

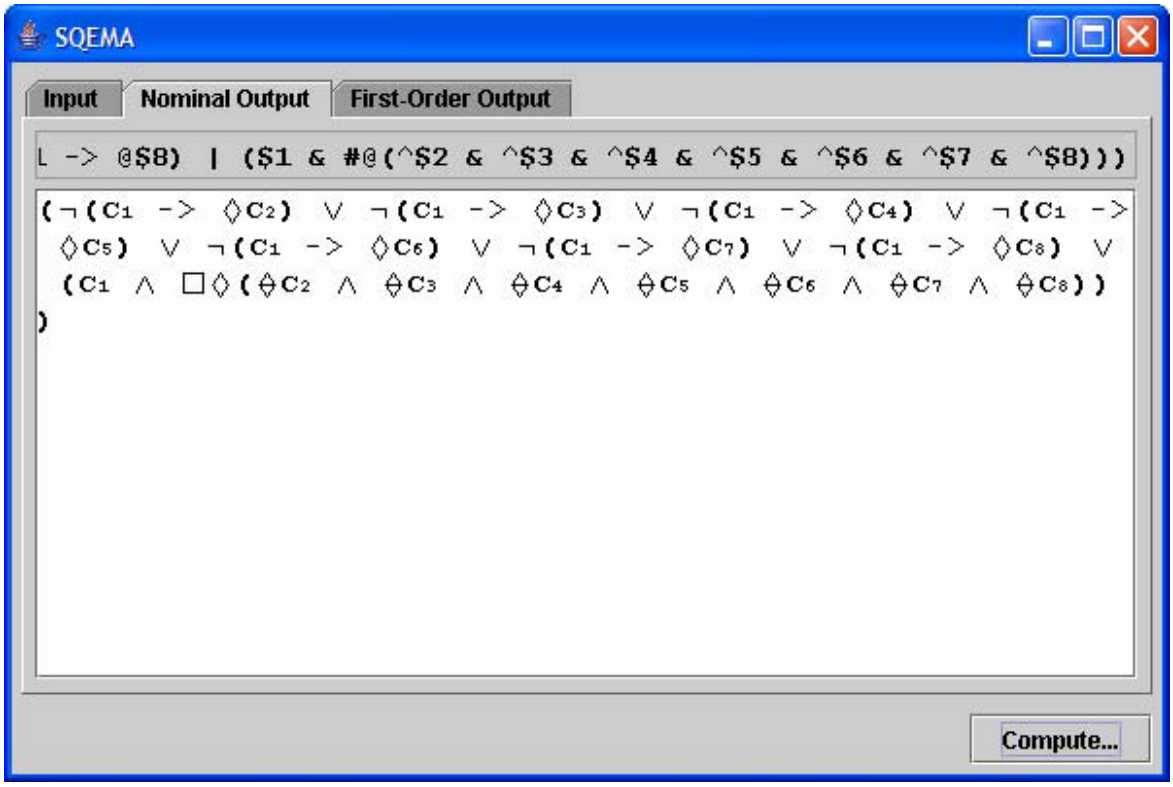
4.1. Swing приложението



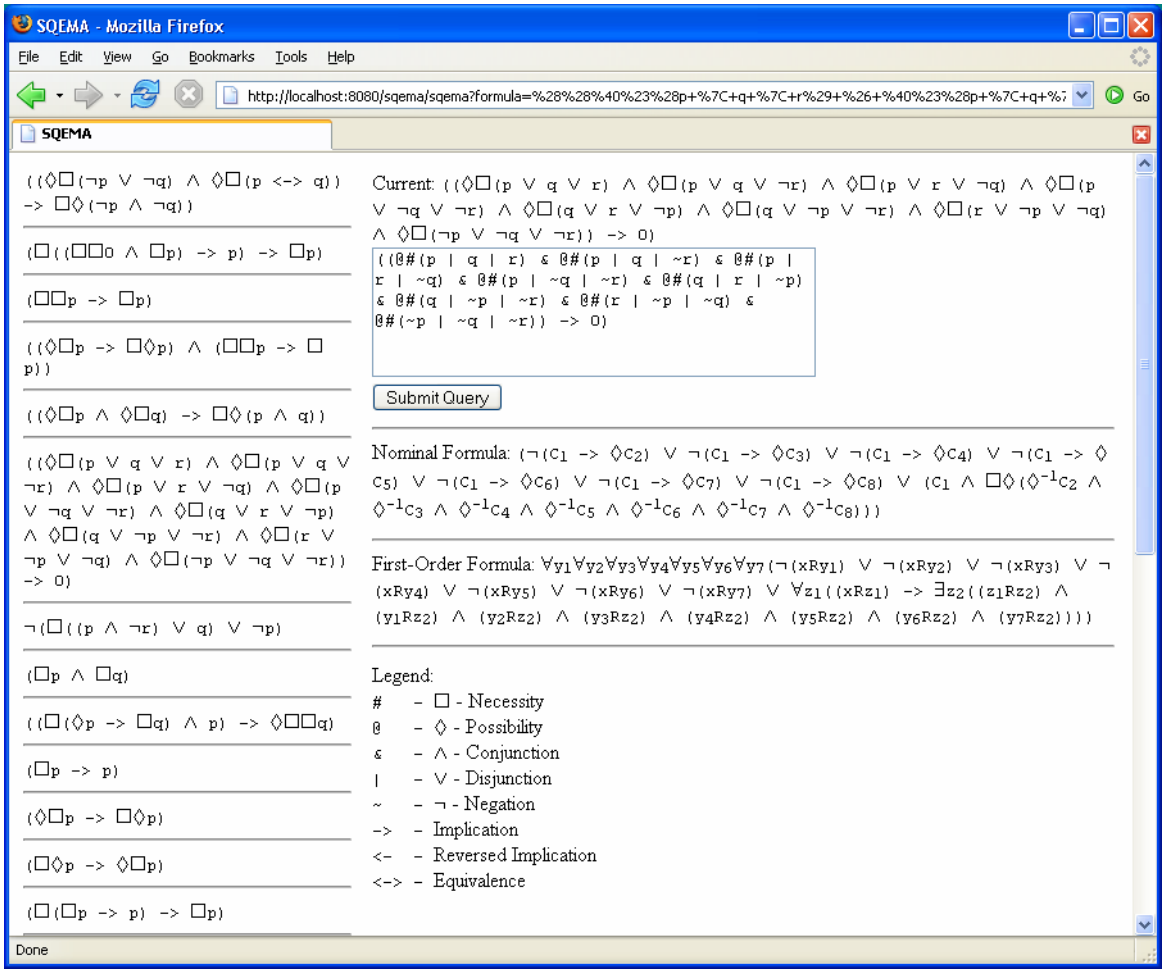
Това е началният екран на приложението. Най-горе се вижда падащо меню с примерни формули. При избиране на някоя от тях, тя се появява в два формата – текстово поле, подлежащо на редактиране, и екран, който изрисува формулата с обичайните логически символи. При натискане на бутона “Compute…” се стартира алгоритъмът SQEMA и резултатът се показва в другите два екрана – Nominal Output и First-Order Output. Фокусът на приложението се сменя и вече се показва третият екран с крайния резултат.



Ако потребителят се интересува от междинния резултат с номинали, той е достъпен от средния екран – Nominal Output.



4.2. Web/Servlet приложението



В това приложение всичко е в един екран. Отдясно има меню с примерни формули, отляво стоят текущата формула (в нормален логически вид и във вид, удобен за редактиране), бутон, стартиращ изчислението, и резултатите, получени в резултат на изчислението. Ако се натисне левият бутон на мишката на елемент от дясното меню, избраната от менюто формула се изчислява автоматично.

5. Заключение

Обяснихме накратко един важен проблем, пронизващ главните линии на изследванията в модалната логика и алгоритъма SQEMA.

Описахме алгоритъма SQEMA на Димитър Вакарелов, Валентин Горанко и Willem Conradie. Обяснихме неговите предимства пред алгоритмите DLS и SCAN.

Обяснихме променената за целите на реализацията версия на алгоритъма и доказахме, че тези промени запазват коректността на алгоритъма.

Обяснихме в детайли системните изисквания на реализацията, както и начина за работа с нея.

Разказахме за всички по-значими части от реализацията и алгоритмите, които се използват в програмата.

Така изпълнихме поставените цели в задачата за реализация на SQEMA.

Използвана литература

[1] Conradie, W., V. Goranko and D. Vakarelov. Algorithmic correspondence and completeness in modal logic I. The core algorithm SQEMA, Logical methods in Computer Science, vol 2 (1:5) 2006, 1-26.

[2] Conradie, W., V. Goranko and D. Vakarelov. Algorithmic correspondence and completeness in modal logic II. Polyadic and hybrid extensions of the algorithm SQEMA. 2006. Submitted.

[3] Gabbay, D. and Ohlbach, H. Quantifier Elimination in Second-order Logic. South African Computer Journal, 7, (1992), 35–43.

[4] Nonnengart, A., H. J. Ohlbach, and A. Szalas. Quantifier elimination for second-order predicate logic, in: Logic, Language and Reasoning: Essays in honour of Dov Gabbay, Part I, H. J. Ohlbach and U. Reyle (eds.), Kluwer, 1997.

[5] A. Szalas, On the correspondence between Modal and Classical Logic: an Automated Approach. J. Logic and Comp. Vol 3 No 6(1993), 605-620.

[6] Ammeraal, L. Algorithms and Data Structures in C++, Hogeschool van Utrecht, Netherlands, 1996, ISBN 0-471-96355-0, ISBN 954-8495-25-2.

[7] Наков, П., Добриков, П., Програмиране = ++Алгоритми;, 2002, ISBN 954-8905-06-X.

ПРИЛОЖЕНИЕ

```
package org.modallogic.algorithm;

import org.modallogic.formula.*;
import org.modallogic.util.set.*;
import org.modallogic.util.sort.*;
import org.modallogic.util.stack.*;
import org.modallogic.util.permutation.*;
import java.util.Enumeration;

////
//// Input:  a modal formula without nominals
//// Output: a modal formula without variables, or null
////
public class SQEMA implements PermutationUser {

  private static boolean Trace = false;
  private static java.io.PrintStream Out = null;

  public static void guiInit() {
    String debugBoolean = System.getProperty("sqema.debug", "false");
    Trace = debugBoolean != null && "true".equals(debugBoolean);
    if (Trace) {
      Out = System.out;
    }
  }

  private static class BacktrackContext {

    public SetHash processed;
    public SetHash remaining;
    public DisjunctionOrdered current;
    public NominalContext nominalContext;
    public int variableIndex;
    public boolean reversePolarity;

    public BacktrackContext(
      SetHash processed,
      SetHash remaining,
      DisjunctionOrdered current,
      NominalContext nominalContext,
      int variableIndex,
      boolean reversePolarity
    ) {
      this.processed = processed.copy();
      this.remaining = remaining.copy();
      this.current = current;
      this.nominalContext = nominalContext.copy();
      this.variableIndex = variableIndex;
      this.reversePolarity = reversePolarity;
    }

    public String toString() {
      return
        "[ processed = " + processed +
        ", remaining = " + remaining +
        ", current = " + current +
        ", nominalContext = " + nominalContext +
        ", variableIndex = " + variableIndex +
        ", reversePolarity = " + reversePolarity +
        " ]"
        ;
    }

  }

  private FormulaPart starting = null;
  private Variable[] variables = null;
  private FormulaPart result = null;
  private NominalContext nominalContext = null;
  private Nominal startingNominal = null;

  public boolean usePermutation(int[] indices) {
    VectorStack backtrack = new VectorStack();
    SetHash processed = new SetHash();
```

```
    SetHash remaining = new SetHash();
    remaining.add(new DisjunctionOrdered(new Not(startingNominal), starting));
    int numVariables = indices.length;
    BacktrackContext context = null;
    for (int variableIndex = 0; variableIndex < numVariables; ++variableIndex) {
      Variable variableToEliminate = variables[indices[variableIndex]];
      Not notVariableToEliminate = new Not(variableToEliminate);
      // Reverse Polarity backtrack point:
      backtrack.push(
        new BacktrackContext(
          processed,
          remaining,
          null,
          nominalContext,
          variableIndex,
          true));
      for (;;) {
        // Restore context, part 1 of 4:
        if (context != null) {
          processed = context.processed;
          remaining = context.remaining;
          nominalContext.set(context.nominalContext);
          variableIndex = context.variableIndex;
          variableToEliminate = variables[indices[variableIndex]];
          notVariableToEliminate = new Not(variableToEliminate);
          if (context.reversePolarity) {
            // Reverse the polarity of the current variable:
            processed = reversePolarity(processed, variableToEliminate,
notVariableToEliminate);
            remaining = reversePolarity(remaining, variableToEliminate,
notVariableToEliminate);
          }
        }
        if (Trace) {
          trace(processed, remaining);
        }
        while (remaining.size() > 0) {
          // 1. Find a formula from the system with this variable:
          FormulaPart toProcess;
          // Restore context, part 2 of 4:
          if (context != null && context.current != null) {
            toProcess = context.current;
          } else {
            toProcess = (FormulaPart)remaining.elements().nextElement();
            remaining.remove(toProcess);
            if (processed.contains(toProcess)) {
              continue;
            }
            if (!findVariable(variableToEliminate, toProcess)) {
              // Finished with the current subformula:
              processed.add(toProcess);
              continue;
            }
            if (
              lemmaAckermannAlpha(toProcess, variableToEliminate,
notVariableToEliminate) ||
              lemmaAckermannBeta(toProcess, variableToEliminate,
notVariableToEliminate)
            ) {
              // Finished with the current subformula:
              processed.add(toProcess);
              continue;
            }
            if (
              DisjunctionCheck.isDisjunction(toProcess) &&
              !findVariable(variableToEliminate, toProcess.getChildAt(1))
            ) {
              toProcess = new DisjunctionOrdered(toProcess.getChildAt(1),
toProcess.getChildAt(0));
              remaining.add(toProcess);
              continue;
            }
          }
          // 2. Save disjunction backtrack info:
          if (
            context == null &&
            DisjunctionCheck.isDisjunction(toProcess) &&
            DisjunctionCheck.isDisjunction(toProcess.getChildAt(1))
          ) {
            FormulaPart child = toProcess.getChildAt(1);
```

```
            if (!findPositive(variableToEliminate, child.getChildAt(0))) {
              // do nothing.
            } else if (!findPositive(variableToEliminate, child.getChildAt(1))) {
              child = new DisjunctionOrdered(child.getChildAt(1),
child.getChildAt(0));
              toProcess = new DisjunctionOrdered(toProcess.getChildAt(0), child);
              remaining.add(toProcess);
              continue;
            } else {
              backtrack.push(
                new BacktrackContext(
                  processed,
                  remaining,
                  // Swap:
                  new DisjunctionOrdered(
                    toProcess.getChildAt(0),
                    new DisjunctionOrdered(
                      child.getChildAt(1),
                      child.getChildAt(0))),
                  nominalContext,
                  variableIndex,
                  false));
            }
          }
          // Restore context, part 3 of 4:
          context = null;
          // 3. Apply the rules to this formula:
          applyRules(nominalContext, toProcess, processed, remaining);
          if (Trace) {
            trace(processed, remaining);
          }
        }
        // Restore context, part 4 of 4:
        context = null;

        // All processing has been done. Now, see if we can apply Ackermann's
lemma:
        boolean ok = true;

        SetHash alphas = new SetHash();
        SetHash betas = new SetHash();
        for (Enumeration enumeration = processed.elements();
enumeration.hasMoreElements(); ) {
          FormulaPart part = (FormulaPart)enumeration.nextElement();
          boolean alpha = lemmaAckermannAlpha(part, variableToEliminate,
notVariableToEliminate);
          boolean beta = lemmaAckermannBeta(part, variableToEliminate,
notVariableToEliminate);
          if ((!alpha) && (!beta)) {
            if (findVariable(variableToEliminate, part)) {
              // Fail on the current subformula:
              ok = false;
              break;
            }
          }
          if (alpha) {
            alphas.add(part);
          }
          if (beta) {
            betas.add(part);
          }
        }

        // Apply Ackermann's lemma:
        // if we have no alphas or no betas, then the next
        // normalization will take care of that variable,
        // replacing it with 0 or 1.
        if (ok && alphas.size() > 0 && betas.size() > 0) {
          processed.removeAll(alphas);
          processed.removeAll(betas);
          FormulaPart alphaConjunction;
          {
            int size = alphas.size();
            FormulaPart[] alphasArray = new FormulaPart[size];
            Enumeration enumeration = alphas.elements();
            for (int alphaIndex = 0; alphaIndex < size; ++alphaIndex) {
              FormulaPart disjunction = (FormulaPart)enumeration.nextElement();
              alphasArray[alphaIndex] =
disjunction.getChildAt(0).equals(variableToEliminate) ?
                disjunction.getChildAt(1) : disjunction.getChildAt(0);
```

```
            }
            alphaConjunction = new ConjunctionMultiple(alphasArray);
          }
          for (Enumeration enumeration = betas.elements();
enumeration.hasMoreElements(); ) {
            FormulaPart beta = (FormulaPart)enumeration.nextElement();
            // Replace the negative variable with alphaConjunction:
            beta = replaceNegative(beta, notVariableToEliminate, alphaConjunction);
            // Store the result as processed:
            processed.add(beta);
          }
        }

        // If Ackermann's rule has been successful, continue with the next
variable:
        if (ok) {
          // Ackermann's rule was successful:
          break;
        }

        // Ackermann's rule has failed:

        // Else, backtrack to the first save point
        // (this could even mean going back a few variables):

        if (backtrack.empty()) {
          // fail this permutation:
          return false;
        }

        // Backtrack:
        context = (BacktrackContext)backtrack.pop();
        continue;
      }

      // Ackermann's rule has been successful for the last variable.
      // We will keep all restore points in case some of the next
      // variables fail.

      // Now, switch remaining and processed:
      {
        SetHash temp = processed;
        processed = remaining;
        remaining = temp;
      }

      if (variableIndex < numVariables - 1) {

        // See if the remaining formulae can be simplified,
        // Also, make them in the form of (A or B).
        // Prefer (not(c) or B) but use (0 or B) if not(c)
        // is not directly available.
        // For now, take no special measures to ensure
        // the presence of not(c).
        // Formula sorting helps somewhat, but this is only heuristics.

        normalize(remaining);

      }

    }

    ConjunctionMultiple resultConjunction = new ConjunctionMultiple(
      (FormulaPart[])remaining.toArray(new FormulaPart[remaining.size()]));
    this.result = normalizeResult(new
Not(CNFToFormula.convert(resultConjunction)));

    return true;
  }

  private static void trace(SetHash a, SetHash b) {
    Out.println("-- Trace --");
    SetHash all = new SetHash();
    all.addAll(a);
    all.addAll(b);
    FormulaPart[] array = new FormulaPart[all.size()];
    all.toArray(array);
    QuickSort.sort(array);
    for (int i = 0; i < array.length; ++i) {
      Out.println(array[i]);
```

```
    }
    Out.println("-- End Trace --");
  }

  // 1. Ackermann condition alpha: (A or p):
  private static boolean lemmaAckermannAlpha(
    FormulaPart formula,
    Variable variableToEliminate,
    Not notVariableToEliminate
  ) {
    if (DisjunctionCheck.isDisjunction(formula)) {
      if (
        formula.getChildAt(1).equals(variableToEliminate) &&
        (!findVariable(variableToEliminate, formula.getChildAt(0)))
      ) {
        return true;
      }
      if (
        formula.getChildAt(0).equals(variableToEliminate) &&
        (!findVariable(variableToEliminate, formula.getChildAt(1)))
      ) {
        return true;
      }
  }
    return false;
  }

  // 2. Ackermann condition beta: (B(not(p))):
  private static boolean lemmaAckermannBeta(
    FormulaPart formula,
    Variable variableToEliminate,
    Not notVariableToEliminate
  ) {
    if (
      findNegative(notVariableToEliminate, formula) &&
      (!findPositive(variableToEliminate, formula))
    ) {
      return true;
    }
    return false;
  }

  public static void applyRules(
    NominalContext nominalContext,
    FormulaPart toProcess,
    SetHash processed,
    SetHash remaining
  ) {
    if (!DisjunctionCheck.isDisjunction(toProcess)) {
      throw new ClassCastException();
    }
    FormulaPart disjunction = toProcess;
    FormulaPart left = disjunction.getChildAt(0);
    FormulaPart right = disjunction.getChildAt(1);
    if (right.is(Possibility.class)) {
      if (left.is(Not.class) && left.getChildAt(0).is(Nominal.class)) {
        Nominal newNominal = new Nominal(nominalContext);
        processed.add(new RelationCouple(left.getChildAt(0), newNominal));
        remaining.add(new DisjunctionOrdered(new Not(newNominal),
right.getChildAt(0)));
      } else {
        processed.add(disjunction);
      }
    } else if (right.is(Necessity.class)) {
      remaining.add(new DisjunctionOrdered(new NecessityReversed(left),
right.getChildAt(0)));
    } else if (right.is(NecessityReversed.class)) {
      remaining.add(new DisjunctionOrdered(new Necessity(left),
right.getChildAt(0)));
    } else if (right.is(Conjunction.class)) {
      remaining.add(new DisjunctionOrdered(left, right.getChildAt(0)));
      remaining.add(new DisjunctionOrdered(left, right.getChildAt(1)));
    } else if (DisjunctionCheck.isDisjunction(right)) {
      FormulaPart newLeft;
      if (left.is(Zero.class)) {
        newLeft = right.getChildAt(0);
      } else {
        newLeft = new DisjunctionOrdered(left, right.getChildAt(0));
      }
      remaining.add(new DisjunctionOrdered(newLeft, right.getChildAt(1)));
```

```
    } else if (right.is(Not.class) || right.is(Variable.class) ||
right.is(Nominal.class)) {
      processed.add(disjunction);
    } else {
      processed.add(normalize(disjunction));
    }
  }

  private static SetHash reversePolarity(
    SetHash system,
    Variable variableToEliminate,
    Not notVariableToEliminate
  ) {
    SetHash result = new SetHash();
    for (Enumeration enumeration = system.elements();
enumeration.hasMoreElements(); ) {
      FormulaPart part = (FormulaPart)enumeration.nextElement();
      result.add(reversePolarity(part, variableToEliminate,
notVariableToEliminate));
    }
    return result;
  }

  private static FormulaPart reversePolarity(
    FormulaPart part,
    Variable var,
    Not notVar
  ) {
    if (
      part.is(RelationCouple.class)||
      part.is(Nominal.class) ||
      part.is(One.class) ||
      part.is(Zero.class)
    ) {
      return part;
    }
    if (part.is(Possibility.class)) {
      return new Possibility(reversePolarity(part.getChildAt(0), var, notVar));
    } else if (part.is(Necessity.class)) {
      return new Necessity(reversePolarity(part.getChildAt(0), var, notVar));
    } else if (part.is(NecessityReversed.class)) {
      return new NecessityReversed(reversePolarity(part.getChildAt(0), var,
notVar));
    } else if (part.is(Conjunction.class)) {
      return new Conjunction(
        reversePolarity(part.getChildAt(0), var, notVar),
        reversePolarity(part.getChildAt(1), var, notVar));
    } else if (part.is(Disjunction.class)) {
      return new Disjunction(
        reversePolarity(part.getChildAt(0), var, notVar),
        reversePolarity(part.getChildAt(1), var, notVar));
    } else if (part.is(DisjunctionOrdered.class)) {
      return new DisjunctionOrdered(
        reversePolarity(part.getChildAt(0), var, notVar),
        reversePolarity(part.getChildAt(1), var, notVar));
    } else if (part.is(Not.class) || part.is(Variable.class) ||
part.is(Nominal.class)) {
      if (var.equals(part)) {
        return notVar;
      }
      if (notVar.equals(part)) {
        return var;
      }
      return part;
    } else {
      throw new RuntimeException("Unknown type: " + part.getClass().getName());
    }
  }

  private static FormulaPart replaceNegative(
    FormulaPart beta,
    Not notVar,
    FormulaPart alphaConjunction
  ) {
    if (
      beta.is(RelationCouple.class)||
      beta.is(Nominal.class) ||
      beta.is(Variable.class) ||
      beta.is(One.class) ||
      beta.is(Zero.class)
```

```
    ) {
      return beta;
    }
    if (beta.is(Possibility.class)) {
      return new Possibility(replaceNegative(beta.getChildAt(0), notVar,
alphaConjunction));
    } else if (beta.is(Necessity.class)) {
      return new Necessity(replaceNegative(beta.getChildAt(0), notVar,
alphaConjunction));
    } else if (beta.is(NecessityReversed.class)) {
      return new NecessityReversed(replaceNegative(beta.getChildAt(0), notVar,
alphaConjunction));
    } else if (beta.is(Conjunction.class)) {
      return new Conjunction(
        replaceNegative(beta.getChildAt(0), notVar, alphaConjunction),
        replaceNegative(beta.getChildAt(1), notVar, alphaConjunction));
    } else if (beta.is(Disjunction.class)) {
      return new Disjunction(
        replaceNegative(beta.getChildAt(0), notVar, alphaConjunction),
        replaceNegative(beta.getChildAt(1), notVar, alphaConjunction));
    } else if (beta.is(DisjunctionOrdered.class)) {
      return new DisjunctionOrdered(
        replaceNegative(beta.getChildAt(0), notVar, alphaConjunction),
        replaceNegative(beta.getChildAt(1), notVar, alphaConjunction));
    } else if (beta.is(Not.class)) {
      if (notVar.equals(beta)) {
        return alphaConjunction;
      }
      return beta;
    } else {
      throw new RuntimeException("Unknown type: " + beta.getClass().getName());
    }
  }

  private static FormulaPart partialSqema(FormulaPart modalFormula, NominalContext
nominalContext, Nominal startingNominal) {
    // negate and prepare the formula:
    modalFormula = normalize(new Not(modalFormula));
    SetHash toEliminate = findToEliminate(modalFormula);
    if (toEliminate.size() < 1) {
      FormulaPart result = normalizeResult(
        new Not(new Disjunction(new Not(startingNominal), modalFormula)));
    }
    SQEMA sqema = new SQEMA();
    sqema.starting = modalFormula;
    sqema.startingNominal = startingNominal;
    sqema.nominalContext = nominalContext;
    sqema.variables = new Variable[toEliminate.size()];
    toEliminate.toArray(sqema.variables);
    QuickSort.sort(sqema.variables);
    Permutations.permutations(toEliminate.size(), sqema);
    return sqema.result;
  }

  public static FormulaPart sqema(FormulaPart modalFormula) {
    NominalContext nominalContext = new NominalContext();
    FormulaPart previous;
    do {
      previous = modalFormula;
      modalFormula = CNFToFormula.convert(modalFormula);
      modalFormula = Cleaner.clean(modalFormula);
      modalFormula = NecessityPropagation.convert(modalFormula);
      modalFormula = PartialCNF.convert(modalFormula, false);
    } while (!previous.equals(modalFormula));
    previous = null;
    SetHash result = new SetHash();
    Nominal startingNominal = new Nominal(nominalContext);
    int size = modalFormula.getNumberOfChildren();
    for (int i = 0; i < size; ++i) {
      FormulaPart partialResult = CNFToFormula.convert(modalFormula.getChildAt(i));
      partialResult = partialSqema(partialResult, nominalContext, startingNominal);
      if (partialResult == null) {
        return null;
      }
      result.add(partialResult);
    }
    ConjunctionMultiple resultConjunction = new ConjunctionMultiple(
      (FormulaPart[])result.toArray(new FormulaPart[result.size()]));
    return resultConjunction;
  }
```

```
private static void normalize(SetHash system) {
  FormulaPart formula = new ConjunctionMultiple(
    (FormulaPart[])system.toArray(new FormulaPart[system.size()]));
  system.clear();
  formula = CNFToFormula.convert(formula);
  formula = normalize(formula);
  if (DisjunctionCheck.isDisjunction(formula)) {
    FormulaPart child = formula.getChildAt(0);
    if (child.is(Not.class) && child.getChildAt(0).is(Nominal.class)) {
      system.add(formula);
      return;
    }
  }
  system.add(new DisjunctionOrdered(Zero.Instance, formula));
}

private static FormulaPart normalize(FormulaPart formula) {
  FormulaPart previous;
  do {
    previous = formula;
    formula = Cleaner.clean(formula);
    formula = PartialCNF.convert(formula, true);
    formula = CNFToFormula.convert(formula);
    SetHash posAndNeg = findToEliminate(formula);
    formula = removeVariables(formula, posAndNeg);
  } while (!formula.equals(previous));
  return formula;
}

private static FormulaPart normalizeResult(FormulaPart formula) {
  FormulaPart previous;
  do {
    previous = formula;
    formula = Cleaner.clean(formula);
    SetHash posAndNeg = findToEliminate(formula);
    formula = removeVariables(formula, posAndNeg);
  } while (!formula.equals(previous));
  return formula;
}

private static FormulaPart removeVariables(FormulaPart formula, SetHash toStay) {
  if (formula.is(RelationCouple.class)|| formula.is(Nominal.class)) {
    return formula;
  }
  if (formula.is(Variable.class)) {
    if (toStay.contains(formula)) {
      return formula;
    }
    return One.Instance;
  }
  if (formula.is(Not.class)) {
    FormulaPart child = formula.getChildAt(0);
    if (child.is(Variable.class) && !(toStay.contains(child))) {
      return One.Instance;
    }
    return formula;
  }
  if (formula.is(Conjunction.class)) {
    return new Conjunction(
      removeVariables(formula.getChildAt(0), toStay),
      removeVariables(formula.getChildAt(1), toStay));
  }
  if (formula.is(Disjunction.class)) {
    return new Disjunction(
      removeVariables(formula.getChildAt(0), toStay),
      removeVariables(formula.getChildAt(1), toStay));
  }
  if (formula.is(DisjunctionOrdered.class)) {
    return new DisjunctionOrdered(
      removeVariables(formula.getChildAt(0), toStay),
      removeVariables(formula.getChildAt(1), toStay));
  }
  if (formula.is(Necessity.class)) {
    return new Necessity(
      removeVariables(formula.getChildAt(0), toStay));
  }
  if (formula.is(Possibility.class)) {
    return new Possibility(
      removeVariables(formula.getChildAt(0), toStay));
```

```
    }
    if (formula.is(NecessityReversed.class)) {
      return new NecessityReversed(
        removeVariables(formula.getChildAt(0), toStay));
    }
    if (formula.is(PossibilityReversed.class)) {
      return new PossibilityReversed(
        removeVariables(formula.getChildAt(0), toStay));
    }
    if (formula.is(One.class) || formula.is(Zero.class)) {
      return formula;
    }
    throw new RuntimeException("Unknown type: " + formula.getClass().getName());
  }

  private static SetHash findToEliminate(FormulaPart part) {
    SetHash positives = new SetHash();
    SetHash negatives = new SetHash();
    collectVariables(part, positives, negatives);
    return positives.intersection(negatives);
  }

  private static void collectVariables(
    FormulaPart part,
    SetHash positives,
    SetHash negatives
  ) {
    if (part.is(Not.class) && part.getChildAt(0).is(Variable.class)) {
      negatives.add(part.getChildAt(0));
    } else if (part.is(Variable.class)) {
      positives.add(part);
    } else {
      int size = part.getNumberOfChildren();
      for (int i = 0; i < size; ++i) {
        collectVariables(part.getChildAt(i), positives, negatives);
      }
    }
  }

  private static boolean findVariable(Variable variable, FormulaPart part) {
    return findPart(variable, part);
  }

  private static boolean findNegative(Not not, FormulaPart part) {
    return findPart(not, part);
  }

  private static boolean findPart(FormulaPart toFind, FormulaPart part) {
    if (toFind.equals(part)) {
      return true;
    }
    int size = part.getNumberOfChildren();
    for (int i = 0; i < size; ++i) {
      if (findPart(toFind, part.getChildAt(i)))  {
        return true;
      }
    }
    return false;
  }

  private static boolean findPositive(Variable variable, FormulaPart part) {
    if (part.is(Not.class)) {
      return false;
    }
    if (variable.equals(part)) {
      return true;
    }
    int size = part.getNumberOfChildren();
    for (int i = 0; i < size; ++i) {
      if (findPositive(variable, part.getChildAt(i)))  {
        return true;
      }
    }
    return false;
  }

}
```