Válasz Dr. Szirmay-Kalos László professzor úr bírálatára

MTA értekezés címe: Modelltranszformációk validálása és alkalmazása

Megköszönöm Dr. Szirmay-Kalos László professzor úrnak az értékes észrevételeit, több ponton is megfogalmazott elismerő szavait. Külön köszönöm az észrevételeit a szabályalapú rendszerek validálásához kapcsolódóan, valamint az értekezés ötödik fejezetéhez (3. téziscsoporthoz) tartozóan. Köszönöm a felmerült kérdéseket és megjegyzéseket melyek a megválaszolásukkal lehetőséget adnak az eredmények néhány további részleteinek bemutatására is.

A dokumentumban dőlt betűtípussal szedve szerepelnek a bírálatból átvett kérdések, majd ezt közvetlenül követik a válaszaim.

---

1. kérdés: *A 3. Fejezet a gráf-transzformáció alapú verifikációs módszereket tárgyalja, az eredményeket pedig az 1. Tézis foglalja össze. A Jelölt összefoglalta, rendszerezte és osztályozta a verifikációs lehetőségeket, amely alapján áttekintette az irodalmi eredményeket, új problémákat vetett fel és kijelölte a jövőben vizsgálható megfeleltetéseket (1.1. tézis). Ez a tárgyalás kétségkívül hasznos a témakör áttekintésére, de hiányérzetem volt az egyes részkérdések megoldását tekintve, ahol azt vártam volna el, hogy a Jelölt egy korábban nem vizsgált problémára konkrét megoldást ad, vagy pedig a korábban vizsgáltra a korábbitól eltérő, valamilyen szempontból jobbat dolgoz ki.*

Ahogy egyre több szoftveralkalmazásra és szoftverrendszerre van szükség, úgy egyre több fejlesztő dolgozik a társadalom és az ipar különféle területein működő szoftver megoldások fejlesztésén. Magának a fejlesztési folyamatnak a támogatását, valamint az alkalmazások üzemeltetését és karbantartását egyszerűbbé, hatékonyabbá tevő módszereket folyamatosan keressük és alkalmazzuk. A modellalapú szoftverfejlesztés során a szoftverrendszert, vagy annak adott komponenseit, modellek segítségével definiáljuk, majd a rendszer forráskódját és további szoftvertermékeket az ún. modellfeldolgozók generálják. A modellfeldolgozók révén a modellalapú szoftverfejlesztési módszerek hatékony lehetőséget jelentenek és eszközöket is kínálnak a fejlesztési folyamat erősítésére.

A kutatómunkám során látható volt, hogy a gráfújraírás alapú rendszerekhez kapcsolódó verifikációs/validációs szempontokat és lehetőségeket támogató módszerek iránt nagy igény mutatkozik. A központi kérdés a következő: mely feltételek verifikálását/validálását tudják a modelltranszformációk biztosítani, valamint mely további feltételek verifikálását/validálását lenne szükséges biztosítaniuk? Ennek megfelelően a terület határait igyekeztem felderíteni. Ehhez megvizsgáltam az elérhető megoldásokat, a terület kutatóival konzultáltam, több a szűk területhez tartozó workshopon (Barbados Graph Theory Workshop 2008, Bellairs Research Institute of McGill University; MODELS konferencia Multi-Paradigm Modeling workshop 2010 és 2011) összefoglaltam az eredményeket, továbbá a konferenciákon felmerült kérdések és javaslatok figyelembevételével dolgoztam tovább rajtuk, részleteztem, majd publikáltam a továbbfejlesztett munkámat. A visszajelzések alapján látható volt, hogy egy viszonylag szűk közösség számára hasznos, de többek számára is értékes megoldás került kidolgozásra. Mivel ezt az irányt hasznosnak ítélte meg a nemzetközi közösség, ezért jelöltem meg tézispontként az értekezésemben is.

Az eredmény lényege azon *modelltranszformációs tulajdonságosztályok* (modelltranszformációk és közvetve a modellek tulajdonságainak halmaza) *bevezetése*, amelyek a modelltranszformációs megközelítések verifikációs és validációs képességeihez tartozó osztályozást támogatni tudják. A tulajdonságosztályok lehetővé teszik a modelltranszformációs megközelítésekkel szemben támasztott elvárások definiálását, valamint meglévő modelltranszformációs eszközök és megközelítések kategóriákba sorolását.

A visszajelzések alapján a tulajdonságosztályok bevezetésének hasznosságában meghatározó erővel bír, hogy (i) támogatják a különböző verifikációs/validációs megközelítések és eszközök összehasonlítását, (ii) segítik a megfelelő verifikációs/validációs megközelítések és eszközök azonosítását egy verifikációt/validációt igénylő feladathoz.

A tulajdonságok egy része független a bemeneti és a cél nyelvektől, ilyen például a terminálás vagy a globális determinisztikusság, mások, mint például a szemantikus tulajdonságok a modellezési (szakterületi) nyelvekhez kötődnek.

Célom volt egyrészt a fentiek szerinti kutatómunka, melynek eredménye a létező megoldásokat a kidolgozott koordinátarendszerben mérni és értékelni tudja. Természetesen annak megmutatása is motivált, hogy a VMTS-ben megvalósításra került eredményeim számos esetben az elvárt funkciókat és azok adott szintjét teljesítik.

---

2. kérdés: *A szabályalapú rendszerek validálása (1.2. tézis) részletesebb bizonyítást igényelne. Nem látszik ugyanis, hogy mi garantálja, hogy ha több szabály is alkalmazható volna, akkor az egyik szabály alkalmazása miért nem sértheti meg a többi szabály feltételeit. Ha pedig minden szabály alkalmazásához az összes kimeneti feltétel ellenőrzése szükséges, akkor eléggé magától értetődő, hogy sohasem kaphatunk feltételt sértő eredményt. Sokat segítene, ha egy konkrét példán követhetnénk a szabályok egymás utáni alkalmazását (a 3.2-3.4. ábrák csak egy lépést mutatnak be).*

A területen végzett kutatómunkám célja az volt, hogy amennyiben egy szabály, illetve szabályok véges számú sorozatának, végrehajtása lefut (nem akad meg a szabály végrehajtása valamelyik előfeltétel miatt vagy a törlés/létrehozás/módosítás műveletek során, illetve az utófeltétel ellenőrzése során), akkor egészen biztosak legyünk abban, hogy az adott szabály, illetve szabálysorozat által definiált elvárt eredmény állt elő. Tehát maga a sikeres lefutás ténye jelentse a biztosítékot az elvárt eredményre. Ez volt a fő célom a dinamikus validálással, valamint az 1.2 tézis keretében dinamikus validálásnak a szabály alapú rendszerekre történő illesztésével.

A szabály alapú rendszer kialakításától függően fennállhat az az eset, amikor egy adott állapotban több szabály is alkalmazható. A munkámban ezzel a változattal nem foglalkoztam. Feltételeztem, hogy a szabály alapú rendszer úgy kerül kialakításra, hogy az induláskor (következő lépés választásakor) választható szabályok bal oldala (feltétel oldala) nem tartalmaz átfedést. A bíráló felvetése ugyanakkor jogos, egybeesik számos, a területen jelenleg folyó kutatási irány célkitűzésével.

Az értekezésben tárgyalt megoldásommal kapcsolatban röviden összefoglalom a következőket. A transzformációs rendszer az én esetemben biztosít vezérlésmodell (control flow) támogatást. Segítségével az újraírási szabályok futása szekvenciába rendezhető, tartalmazhat feltételes elágazásokat (mintaillesztés alapján vagy attribútum értékek kiértékelése alapján), valamint

ciklikus szerkezeteket. Szabály alapú rendszerek esetén hasonlóan, egy kezdeti feltétel alapján (legtöbb esetben egy meglévő struktúra esetén) indul el a feldolgozás más-más ágon, de ott több egymást követő lépés végrehajtása történik szigorúan definiált sorrendben. Ez több külön transzformációs vezérlésmodellt jelent, ahol az első szabály bal oldala (struktúra és a kényszerek formájában definiált további előfeltételek) jelenti a kiválasztási szempontot.

A transzformációs szabályok jobb oldalához rendelt, szintén kényszerek formájában definiált, utófeltételek ellenőrzésre kerülnek a strukturális módosítások elvégzését követően. A létrehozott új gráfstruktúra attribútum értékeit csak ezen utófeltételek ellenőrizhetik, de navigációs lépések révén ki is tudnak lépni a szabály jobb oldala által lefedett struktúrából és a generált/módosított gráfstruktúra környezetében szereplő távolabbi csomópontokra elnavigálnak és azoknak a szomszéd vagy még távolabbi csomópontoknak az attribútum értékeit figyelembe tudják venni a sikeres szabály végrehajtás ellenőrzésében. Ez azt jelenti, hogy a szabályok végrehajtásának utófeltételeivel az is ellenőrizhető, hogy megfelelő környezetbe és a megfelelő módon illeszkedik-e a generált/módosított struktúra.

Gyakran nem egy-egy szabállyal, hanem szabályok végrehajtásának sorozatával (teljes transzformációval) szemben kívánunk elvárásokat megfogalmazni. Az 1.2 tézisben ilyen elvárások kezelésére dolgoztam ki azt a módszert, amelyben szabályalapú rendszerek transzformációs szabályok véges számú sorozatának végrehajtásával valósíthatók meg (szélsőséges esetben a sorozat egyetlen szabály végrehajtását is jelentheti) és validáló kényszerekkel (szabályokhoz rendelt elő- és utófeltételekkel) specifikálható a sikeres lefutással kapcsolatos elvárás. Amennyiben ez a szabálysorozat a bemeneti modellre alkalmazva sikeresen végrehajtásra kerül, akkor a módosított/kimeneti modell megfelel a transzformációs szabályokban a validáló kényszerek által definiált elvárásoknak.

A bíráló felvetésére, kifejezett kérésére egy példát mellékelek, amely egy hosszabb modelltranszformációt mutat be teljes részletességgel, és amelyen jól követhető a szabályok egymás utáni alkalmazása. Ezt a példát azért választottam, mert itt a hagyományos jelölésrendszert követik a transzformációs szabályok (külön ábrázolva a szabály bal oldala és a jobb oldala). Az értekezésben a 3.4 ábra a kompakt jelölést használja, ahol színek szemléltetik a törlés/létrehozás/módosítás műveleteket. A rendszer felhasználóinak a kompakt jelölés egyszerűsítés, ám egy-egy példa áttekintése esetén a hagyományos jelölésrendszer választása a követhetőséget szolgálja. Következtetésképpen a példa leírása hosszú, ugyanakkor már az első néhány egymást követő szabály alapján látszik, hogy egy transzformációs szabályon belül hogyan zajlódnak le a törlés/létrehozás/módosítás műveletek (internal causality ábrák, a példában közvetlenül a transzformációs szabályok alatt találhatók). Az is látható, hogyan kapcsolódnak egymáshoz a szabályok a végrehajtás során (external causality ábrák, a példában a végén egyben kigyűjtve találhatók).

---

3. kérdés: *A transzformáció terminálásával kapcsolatban megkérdezem, hogy miként viszonyul ez a probléma a számításelmélet megállási problémájára (lehetetlen olyan programot írni, amely egy tetszőleges programról és tetszőleges bemenetéről véges lépésben eldönti, hogy az erre a bemenetre végtelen ideig fut-e vagy sem).*

A gráfújrairási rendszerek esetén a terminálás kérdése általános esetben nem eldönthető. Az ezzel kapcsolatos vizsgálódások kiindulási alapját a következő publikáció jelenti:

D. Plump, Termination of graph rewriting is undecidable, In Fundam. Inf., Vol. 33(2), Amsterdam, The Netherlands: IOS Press, pp. 201-209, 1998.

A cikk absztraktja angol nyelven:

„Abstract. It is shown that it is undecidable in general whether a graph rewriting system (in the "double pushout approach") is terminating. The proof is by a reduction of the Post Correspondence Problem. It is also argued that there is no straightforward reduction of the halting problem for Turing machines or of the termination problem for string rewriting systems to the present problem."

Fentiek értelmében általánosságban egy gráfújraírási rendszer (esetemben a "double pushout approach" (DPO) alapú gráfújraírási rendszerek) esetén nincs olyan mechanizmus, amely lehetővé tenné a terminálással kapcsolatos biztos és automatizált döntést. Kutatók különböző terminálási kritériumokat (kritérium rendszereket) dolgoztak ki a területen. Ezek a megközelítések az újraírási szabályok végrehajtási sorrendjéhez kapcsolódó rétegelt nyelvtanokon (layered grammars) vagy egyéb mérhető funkciókon/képességeken alapulnak. Segítségükkel a terminálást megfelelő kritériumok meglétével tudjuk igazolni. Ezen megoldásokból néhány legjobban kapcsolódót mutat be az értekezés *3.3.2.1 Termination of Model Transformations* fejezete.

A terminálási feltételek ellenőrzéséhez képest a transzformáció determinisztikus viselkedése még nehezebb feladat tud lenni. A legtöbb transzformációs rendszer vezérlésmodell (control flow) támogatást biztosít. Néhány transzformációs megközelítés lehetővé teszi a nemdeterminisztikus szabályválasztást. Ez azt jelenti, hogy a transzformáció eredménye eltérő lehet, még akkor is, ha a bemeneti modell ugyanaz. Ezzel a kérdéssel az értekezés *3.3.2.2 Global Determinism of Model Transformations* fejezete foglalkozik.

---

4. kérdés: *A Jelölt a követelmények kezelésére szakterület-specifikus nyelvet, azaz metamodellt javasol, amit UML diagramokkal definiál, és Eclipse keretben implementál. A rendszer értékes és a gyakorlatban is alkalmazható eredmény, aminek hasznát jó lett volna egy konkrét példa segítségével illusztrálni. A releváns tesztek generálása jó, de annak konkrét megvalósítási algoritmusa nem kellően részletesen leírt, pl. a döntési pontokat a rendszer automatikusan választja-e ki, és ha igen, mi alapján, vagy pedig a felhasználótól várja.*

Az összetett szoftverrendszerek összköltségének jelentős részét a tesztelésre és hibajavításra fordítjuk. Az egyre komplexebbé és egymással integráltabbá váló szoftverek szükségessé teszik, hogy a tesztelést minél nagyobb mértékű automatizmussal támogassuk, valamint, hogy a fejlesztéseknek minél korábbi szakaszaiban derítsük fel a hibákat. Az automatikus tesztelés megvalósításával a modell, vagy kód változásakor nem szükséges a teszteseteket manuálisan karbantartani, nincs külön tesztтervezési költség, továbbá jól mérhető a tesztek által lefedett programrészek nagysága is. *(a hatékonyságjavulás 1. komponense)*

Ha modell definiálja az elvárt funkcionalitást, akkor lehetőség van modellalapú tesztelési megközelítéseket használni. Ilyenkor a tesztesetek közvetlenül a modellből származnak. A szakterületi modell által leírt logika tesztelésével már a tervezési fázisban felismerhetjük a hibák egy részét. *(a hatékonyságjavulás 2. komponense)*

A valóban hasznos (releváns) tesztforgatókönyvek generálását biztosító kutatómunka célja azon automatizmusok kidolgozása volt, amelyek a követelményspecifikációt leíró modellek által meghatározott rendszer funkcionalitását teljes mértékben lefedő tesztforgatókönyveket generálnak. Jelentős hatékonyságnövekedést jelent, hogy az esetlegesen igen nagy számban megjelenő irreleváns, valós környezetben elő nem forduló esetek, nem foglalnak tesztelési kapacitást. A tesztesetek állapotterének egyszerűsítése lehetővé teszi az összes releváns forgatókönyv ellenőrzését idő és költséghatékony módon. *(a hatékonyságjavulás 3. komponense)*

A tesztforgatókönyvek generálásának célja, hogy a teszt modellekben, aktivitás diagramokon megtervezett teszt folyamatokból konkrét manuális tesztek elvégését lehetővé tevő, olvasható teszteset leírásokat készítsünk. A teszt tervezés szempontjából lényegesen biztonságosabb, és egyszerűbb is, az összes elvi lehetőségből a szükségteleneket elhagyni, mint minden valóban lényeges kombinációt külön nyilván tartani.

A tesztelendő rendszerek összetett üzleti folyamatokat támogatnak. Egy összetett folyamat általában több részfunkció egymás utáni alkalmazásával áll elő. A részfolyamatok önmagukban is értelmezhetők, vizsgálhatók és a rendszerrel szemben támasztott követelmények is eszerint strukturáltak, ezek a use case-ek. A funkcionális tesztelés alapvetően a use case-ek által leírt folyamatoknak a végig vitelét jelenti. A use case-ek leírása a felhasználói követelményeket fogalmazza meg, a tesztnek ezzel szemben a fejlesztési igényeket kell kielégítenie, így a teszt egyértelműbb leírást igényel. A fő különbség, hogy a tesztben nem lehetnek szabadon felcserélhető sorrendű részfolyamatok.

A use case-ek lehetséges lefutásait folyamatábrákon (aktivitás diagramokon) ábrázoljuk. A rendszerfunkciókat több folyamatban is használhatjuk, ennek megfelelően a hozzá tartozó folyamat modelleket is újra felhasználjuk. Összetett folyamatok modelljét a részfolyamatok modelljeiből állítjuk össze. A módszer alkalmazásának fontos következménye, hogy a részfolyamatokból történő építkezés igazolja azt is, hogy a rendszer specifikációját jelentő use-case-ek alkalmasak-e a kívánt üzleti folyamatok megvalósítására.

A tesztesetek kiindulási alapjai a követelményspecifikáció során elkészített aktivitás diagramok. A modellekkel kapcsolatos elvárások (a modellező környezet ellenőrzi):

- Az aktivitás diagramok vagy use case-hez (konkrét rendszer funkció tesztelési folyamata) kapcsolódnak, vagy csomaghoz (általánosabb tesztelési folyamat).
- Egy modellen belül csak egy diagramot kezelünk (egyetlen *Start* pont lehet).
- A diagramok tetszőleges mélységig egymásba ágyazhatók (újra felhasználható diagram komponensek alkalmazhatók).
- A követelményspecifikáció során készített aktivitás diagramok hiányában csak a tesztelés kedvéért is készíthetünk diagramokat.
- Egy részletes teszt diagramon két úszósáv szerepel: az első a kezdeményező (tesztelési lépést elvégző) felhasználó, a második a tesztelés alatt álló rendszer. Ezek jelennek meg a tesztforgatókönyvben, mint a végrehajtó felhasználó és a tesztelt rendszer.

Az alapértelmezett tesztesetgenerátor logika bejárja az aktivitás diagramot, az elágazások mentén annyi tesztesetet generál, ahány lehetséges lefutása lehet az adott folyamatnak. Az aktivitások, mint ellenőrzési feladatok jelennek meg.

A releváns tesztesetek generálásához két technikát is alkalmazunk:

- Lehetőség van arra, hogy a tesztforgatókönyvben konkrét teszt adatokat jelenítsünk meg anélkül, hogy ezt a modellbe beégetnénk. Generálható egy üres paraméter adatlista. A feldolgozó bejárja az aktivitás diagramot összegyűjti a teszt adat tag-eket és annyi konkrét érték megadását teszi lehetővé, ahány lehetséges bejárás lehet a diagram elágazásai alapján. A generált paraméter adatlista kitöltésekor megadhatunk konkrét értéket (szám, szöveg) vagy intervallum értéket, illetve jelölhető, hogy az attribútum alapértelmezett értékét használja. Amikor a tesztforgatókönyvek generálásánál bemenetként megadjuk a kitöltött paraméter adatlistát, akkor csak azok a tesztesetek kerülnek legenerálásra, ahol a megadott paraméter értékek teljesítik a modellhez tartozó kényszereket/kritériumokat.

- A modellben az elágazásokat csoportokba sorolhatjuk azzal a céllal, hogy elkerüljük az ellentmondásos elágazásokat. Csak azok a tesztesetek lesznek relevánsak és generálódnak le, amelyek útvonalában egy csoporthoz mindenhol ugyanaz az érték tartozik. (például 'Szállítás típusa' = 'Közút' érték esetén semmiféle más ('Vasút', 'Vízi', 'Légi' szállítás opciókhoz tartozó ág nem kerül bejárásra)

Alapértelmezett esetben egy felhasználó kommunikál egy rendszerrel. Természetes igény, hogy legyenek integrációs tesztek, ahol több rendszeren átívelő folyamatokat tudunk tesztelni, többféle felhasználói szerepkörrel. Ezesetben külön diagram szerkeszti egybe a folyamatban egymást követő tesztdiagramokat. A generálás során egységesen kezeljük a teszt adatokat, tehát a tesztelt rendszerek között a folyamat végig vitelével a teszt adatok értékei is továbbadásra kerülnek.

Összefoglalva a kutatómunka eredménye egy olyan eljárás, amelynek segítségével a felhasználói forgatókönyvek alapján a forgatókönyvekhez tartozó valós teszteset leírások állíthatók elő. Ezt a fent leírt módszerek alkalmazása teszi lehetővé. Tapasztalatom szerint a válasz elején bemutatott három faktor, hatékonyságjavulási komponens együttes hatása akár a harmadára/negyedére csökkenti a tesztelési feladatok elvégzéséhez szükséges időt és erőforrásokat.

Ismét megköszönöm a Bíráló kérdéseit és észrevételeit, valamint a bírálat elkészítésére szánt idejét, eredményeim diszkusszióját. Megtisztelő számomra, hogy értékesnek tartja az értekezésben bemutatott eredményeket. Bízom benne, hogy a fentiekben megnyugtató válaszokat tudtam adni a felmerült kérdésekre.

Budapest, 2019. június 10.

Lengyel László

Model Transformations in Practice -- Class model to RDBMS transformation challange

The model transformation challenge is a slight variation on the well known 'class to RDBMS' transformation. This example has been chosen because despite its relative simplicity, it tends to exercise a broad class of model transformation features. First we introduce the challenge itself and then we discuss the solution worked out in the VMTS framework.

9.1.1 Meta-models

The meta-model for class models is shown in figure 1. The following OCL constraint is also part of the model:

```
context Class inv:
  attributes->size > 0 and
  attributes->exists(attr | attr.is_primary = true)
```

A model consists of classes and directed associations. A class consists of one or more attributes, at least one of which must be marked as constituting the classes' primary key. An attribute type is either that of another user class, or of a primitive data type (e.g. String, Int). Associations are considered to have a `1` multiplicity on their destination. Submissions may assume the presence of standard data-types as instances of the `PrimitiveDataType` class.

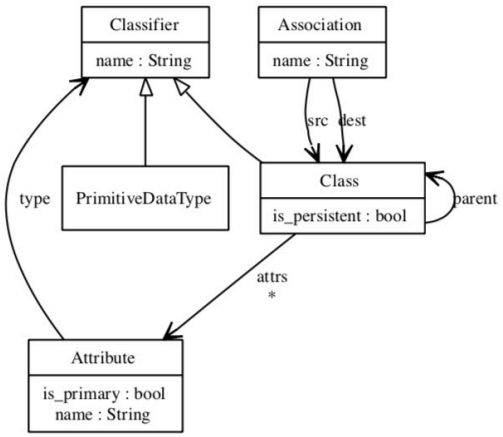


Figure 1: Class meta-model.

The meta-model for RDBMS models is shown in figure 2. An RDBMS model consists of one or more tables. A table consists of one or more columns. One or more of these columns will be included in the `pkey` slot, denoting that the column forms part of the tables primary key slot. A table may also contain zero or more foreign keys. Each foreign key refers to the particular table it identifies, and denotes one or more columns in the table as being part of the foreign key.

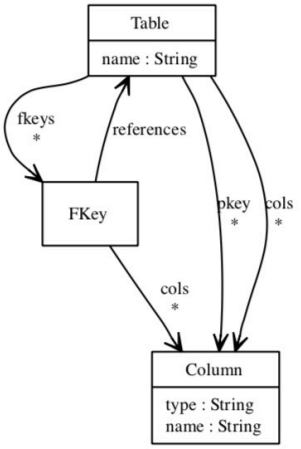


Figure 2: RDBMS meta-model.

9.1.2 Transformation

This version of the transformation contains several subtleties that authors will need to be aware of. In order to facilitate comparisons between approaches, authors should ensure that they accurately implement the transformation.

1. Classes that are marked as persistent in the source model should be transformed into a single table of the same name in the target model. The resultant table should contain one or more columns for every attribute in the class, and one or more columns for every association for which the class is marked as being the source. Attributes should be transformed as per rules 3 – 5.
2. Classes that are marked as non-persistent should not be transformed at the top level. For each attribute whose type is a non-persistent class, or for each association whose `dst` is such a class, each of the classes' attributes should be transformed as per rule 3. The columns should be named *`name_transformed attr`* where *`name`* is the name of the attribute or association in question, and *transformed attr* is a transformed attribute, the two being separated by an underscore character. The columns will be placed in tables created from persistent classes.
3. Attributes whose type is a primitive data type (e.g. String, Int) should be transformed to a single column whose type is the same as the primitive data type.
4. Attributes whose type is a persistent class should be transformed to one or more columns, which should be created from the persistent classes' primary key attributes. The columns should be named *`name_transformed attr`* where *`name`* is the attributes' name. The resultant columns should be marked as constituting a foreign key; the `FKey` element created should refer to the table created from the persistent class.
5. Attributes whose type is a non-persistent class should be transformed to one or more columns, as per rule 2. Note that the primary keys and foreign keys of the translated non-persistent class need to be merged in appropriately, taking into consideration that the translated non-persistent class may contain primary and foreign keys from an arbitrary number of other translated classes.
6. When transforming a class, all attributes of its parent classes (which must be recursively calculated), and all associations which have such classes as a `src`, should be considered. Attributes in subclasses with the same name as an attribute in a parent class are considered to override the parent attribute.
7. In inheritance hierarchies, only the top-most parent class should be converted into a table; the resultant table should however contain the merged columns from all of its subclasses.

Notes on the transformation:

- Rules 2, 4 and 5 are recursive – the 'drilling down' into attributes' types can occur to an arbitrary level.
- Associations do not directly transform into elements; however each association which has a particular class as a `src` must be considered when transforming that class into a table and / or columns.
- When merging the transformation of a non-persistent class, care must be taken to handle the primary and foreign keys of the transformed class appropriately.
- Foreign keys, primary keys and so on should point to the correct model elements – transformations which create duplicate elements with the same names are not considered to provide an adequate solution.

Authors are encouraged to take particular note of the following points when they create their transformations:

- The recursive nature of the drilling down.
- The creation of foreign keys.
- Associations.

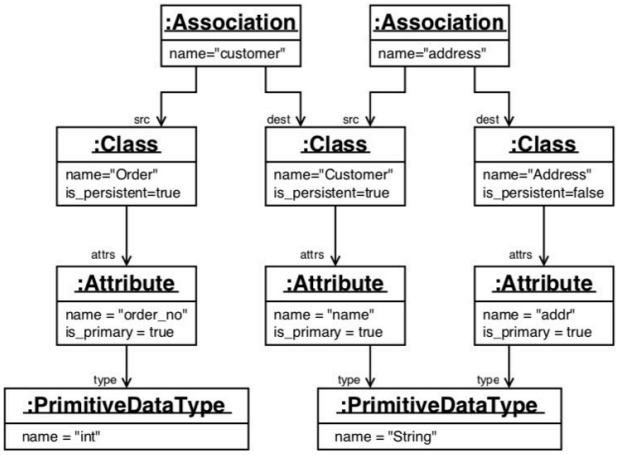


Figure 3: Example input.

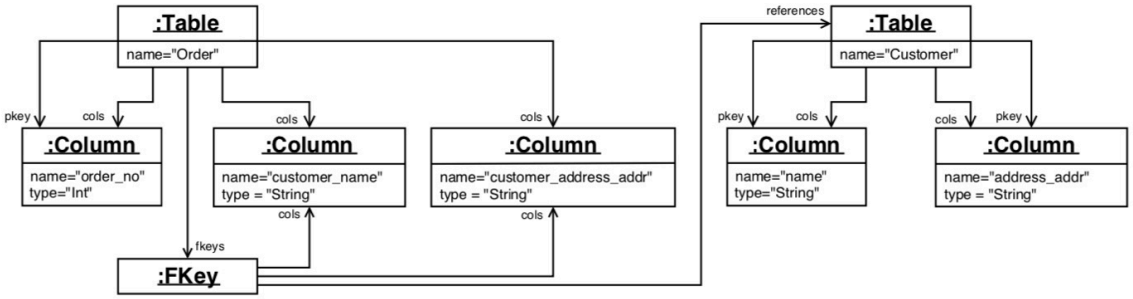


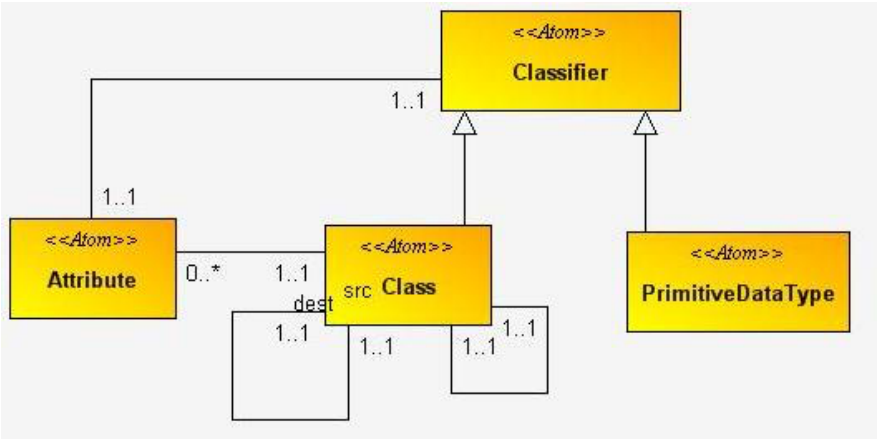
Figure 4: Example output.

9.1.3 Example execution

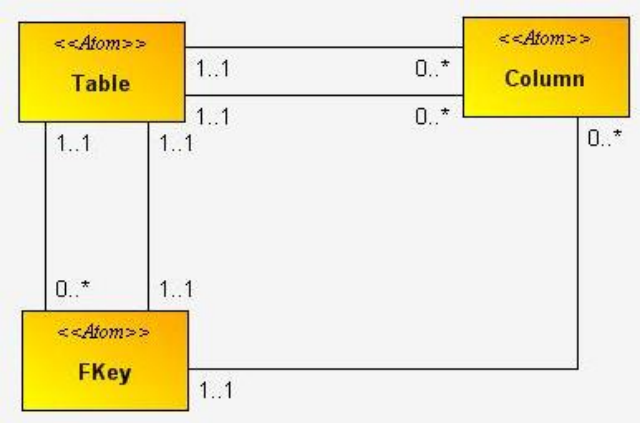
Figures 3 and 4 show the example input and output to the class to RDBMS transformation example.

The solution developed in VMTS framework

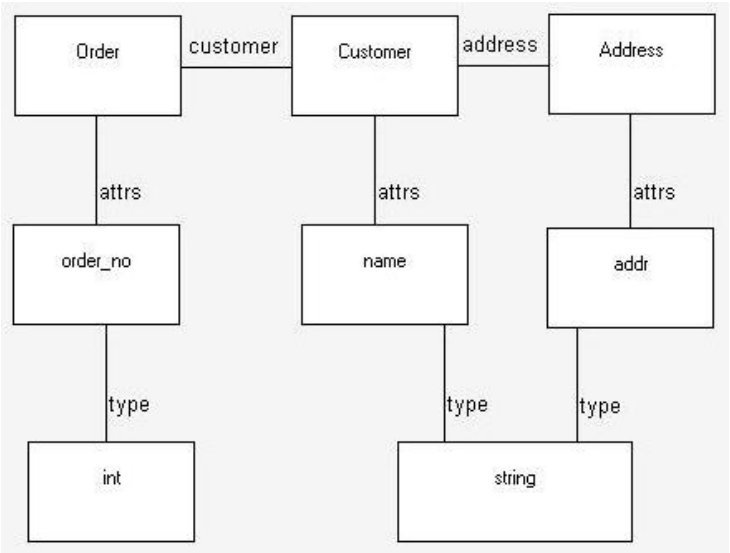
Class metamodel in VMTS:



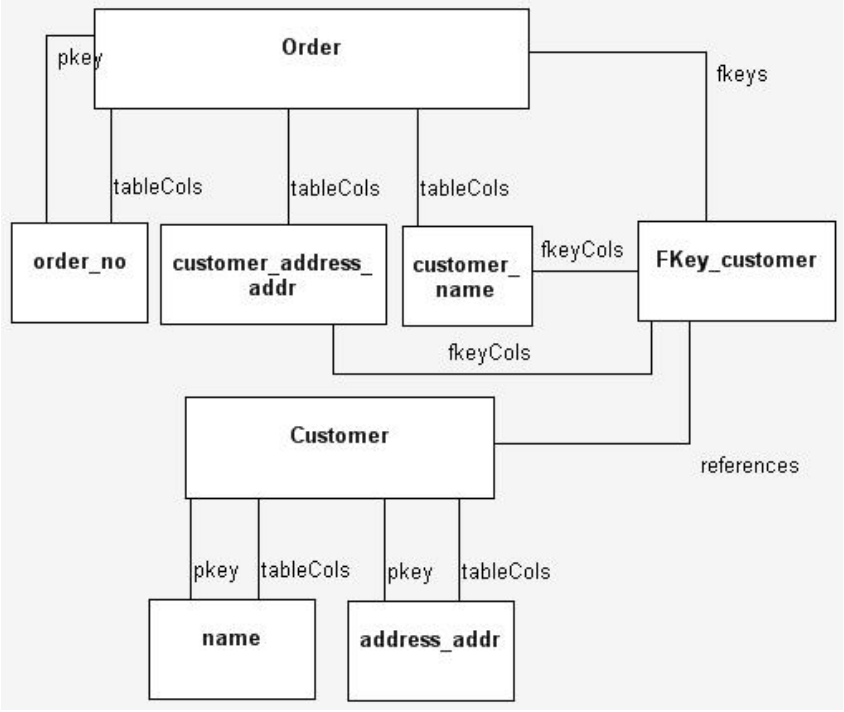
Database metamodel in VMTS:



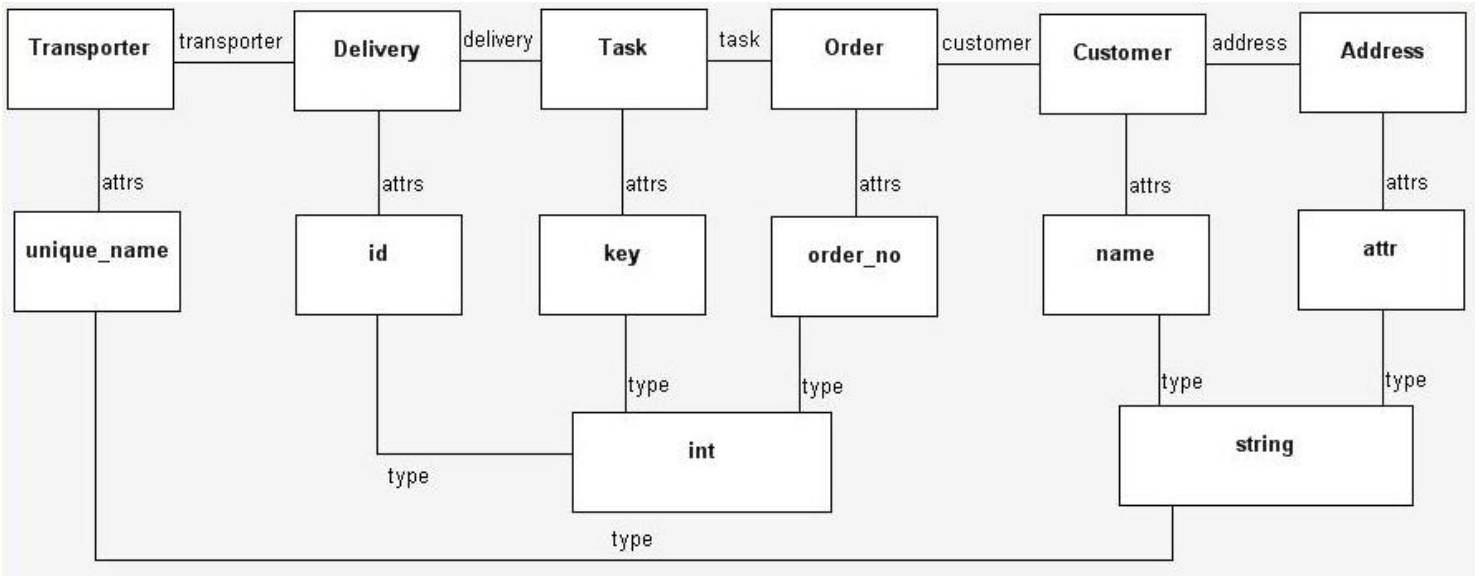
Example input model 1:



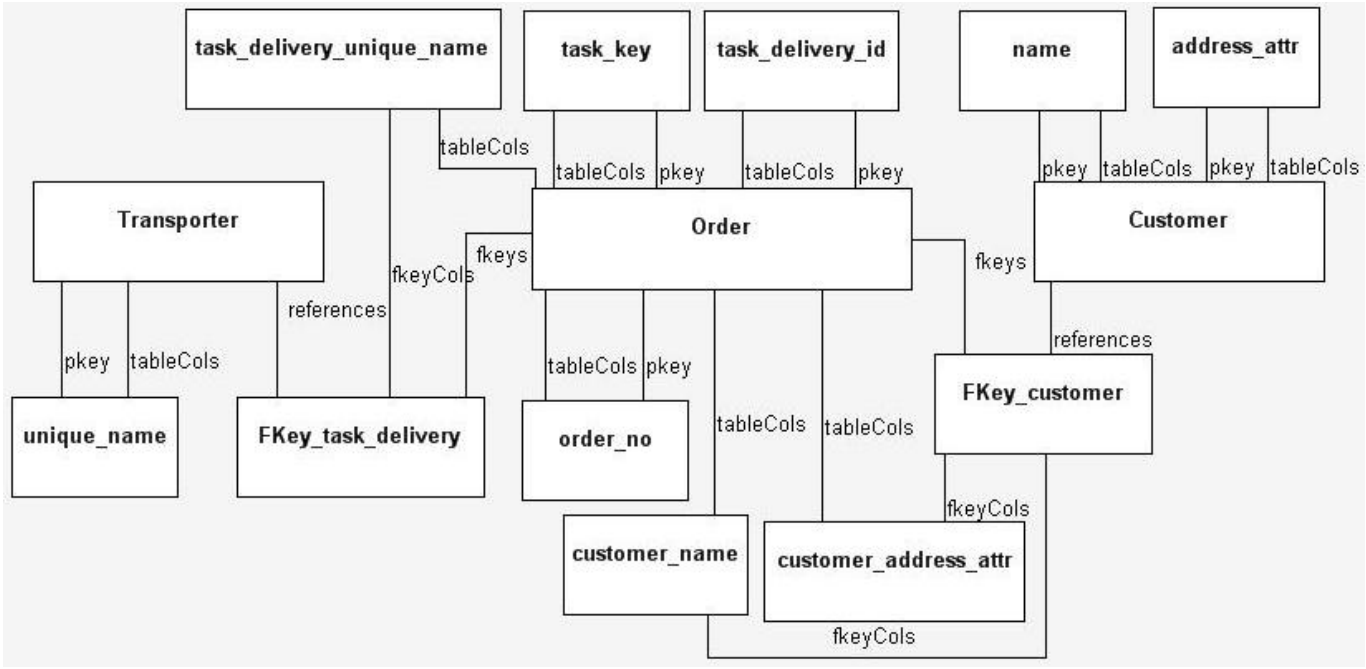
Example output model 1:



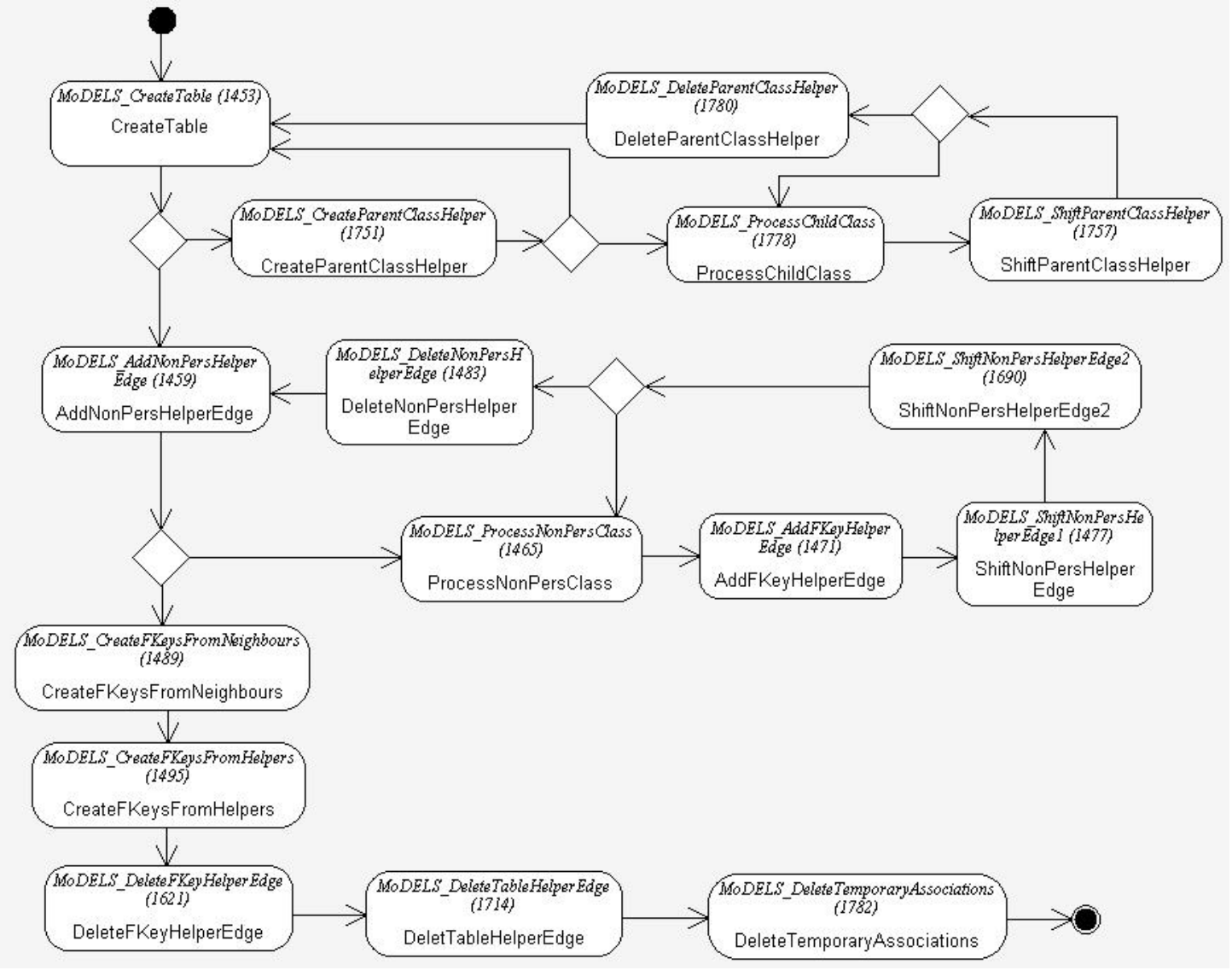
Example input model 2:



Example output model 2:



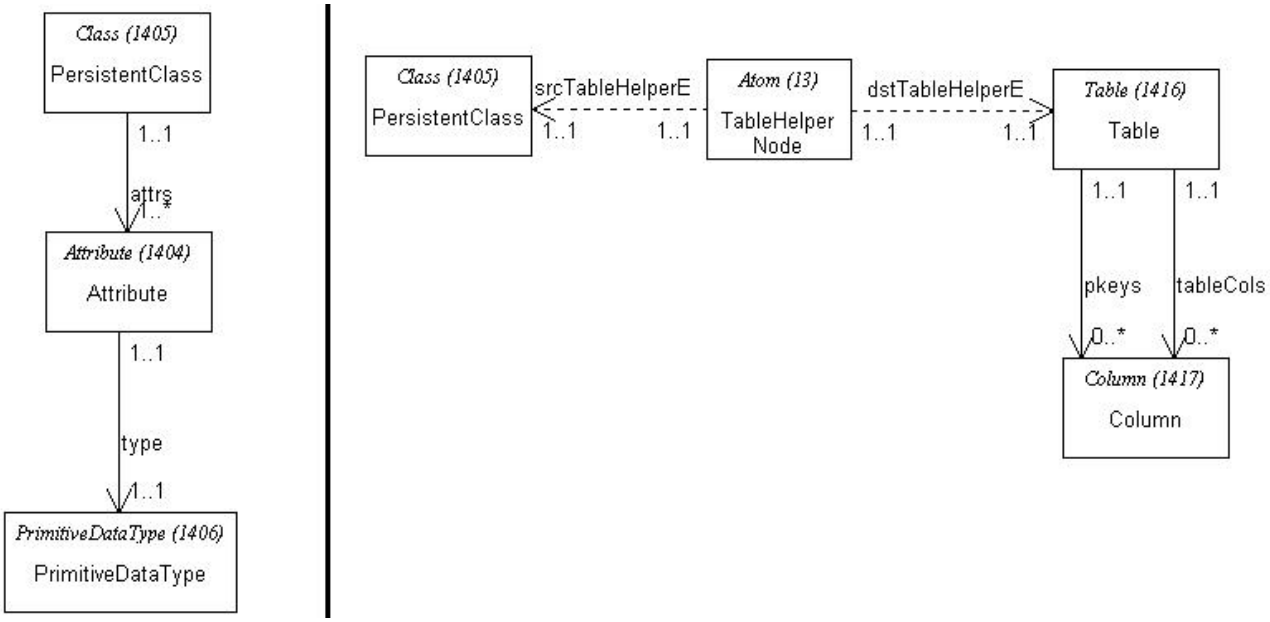
The control flow model of the model transformation. Each node represents a transformation rule and the arrows define the sequence of the transformation process. Also there are conditional branches (in this version of the transformation the branching is decided based on the result of the previous rule execution, i.e. the applicability of the previous rule).



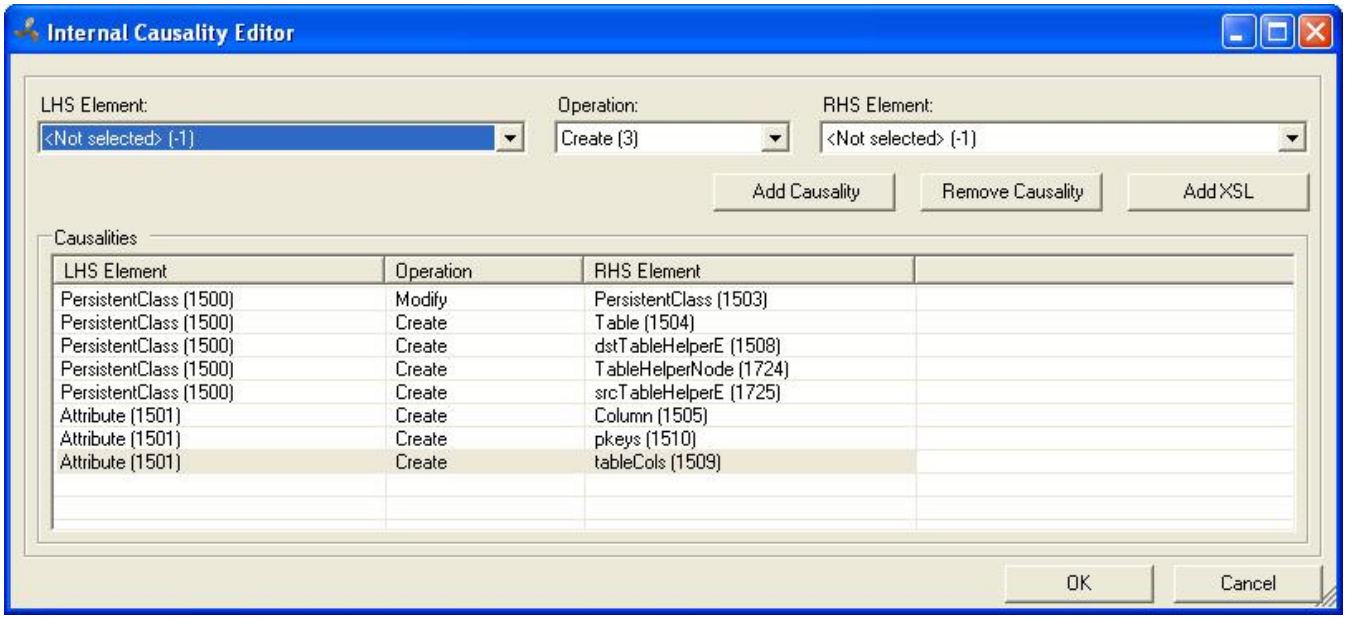
Transformation steps and Internal Causalities

The rules (*CreateTable, CreateParentClassHelper, ProcessChildClass, ShiftParentClassHelper, DeleteParentClassHelper*) are responsible for treating tables and the inheritance-related issues.

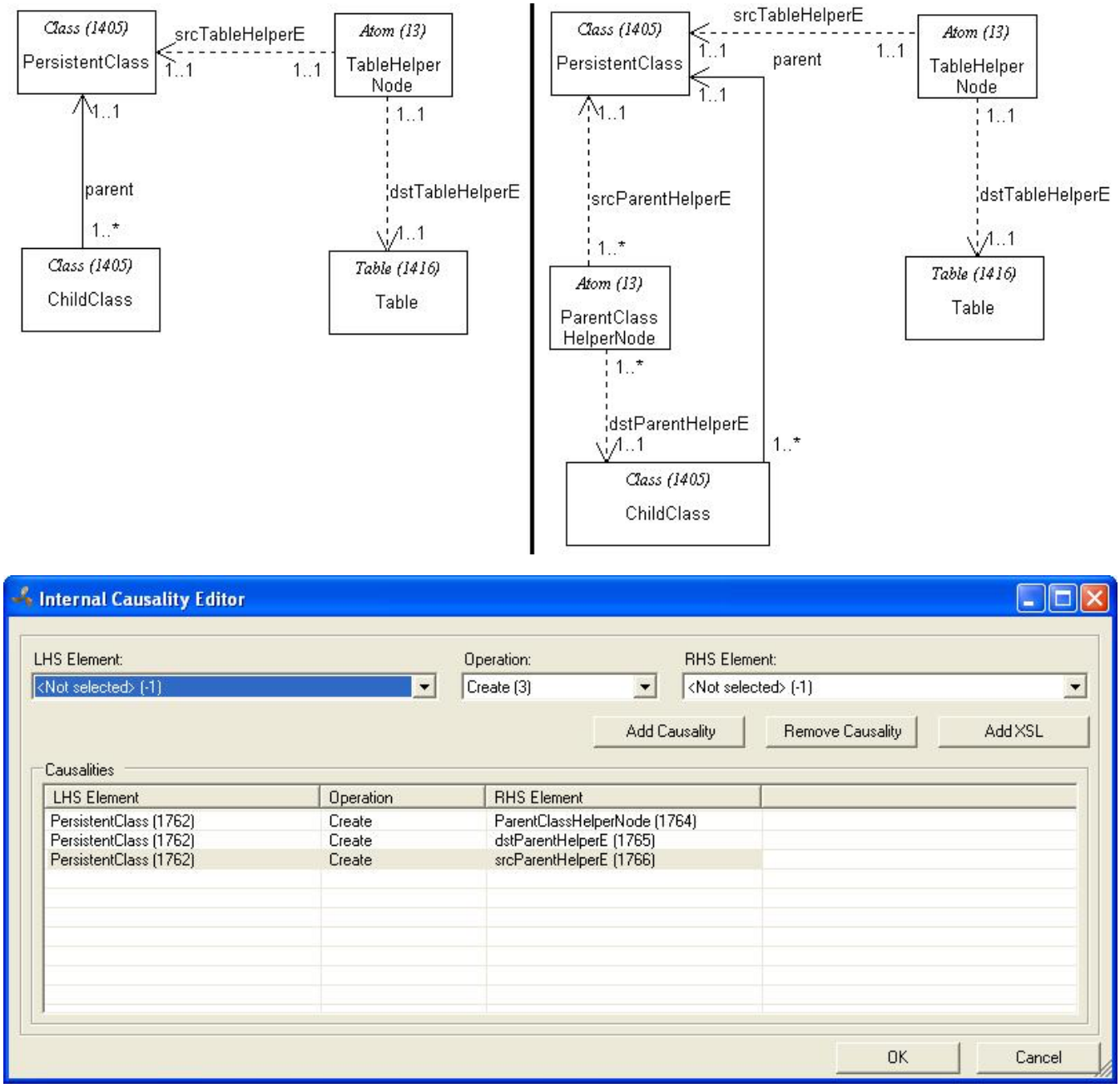
Rule *CreateTable*
The rule processes classes and creates tables into the output model. A resultant table has the same names as the class in the input model. The table contains one added primary key column, and one column for each attribute in the processed class. Furthermore, this rule links each class to their created table. Using this link, the following rules can reach the appropriate table from the class to add new columns and foreign keys to it.



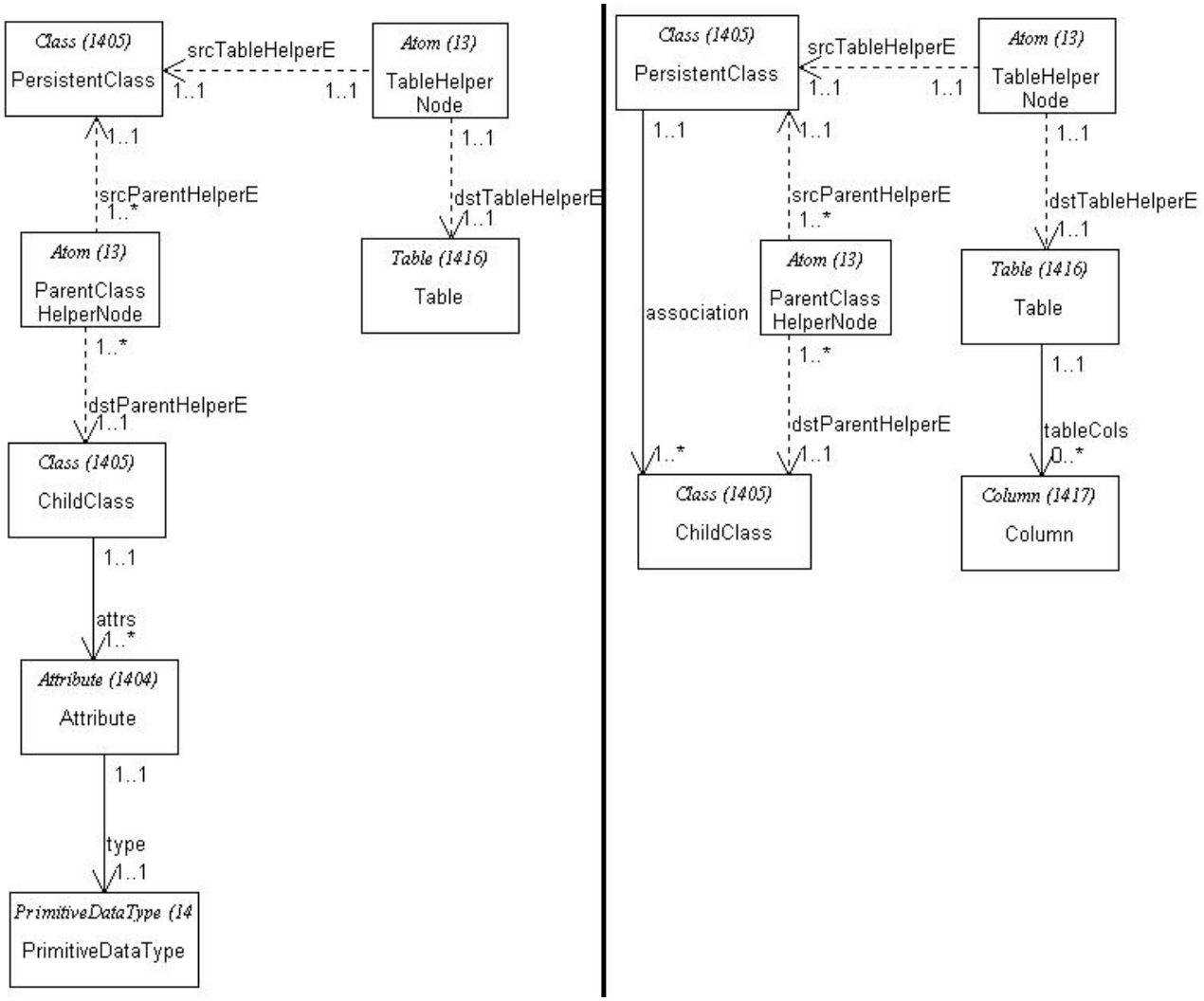
Internal causalities define the Delete/Create/Modify relations between the left-hand side and right-hand side nodes of the transformation rule.

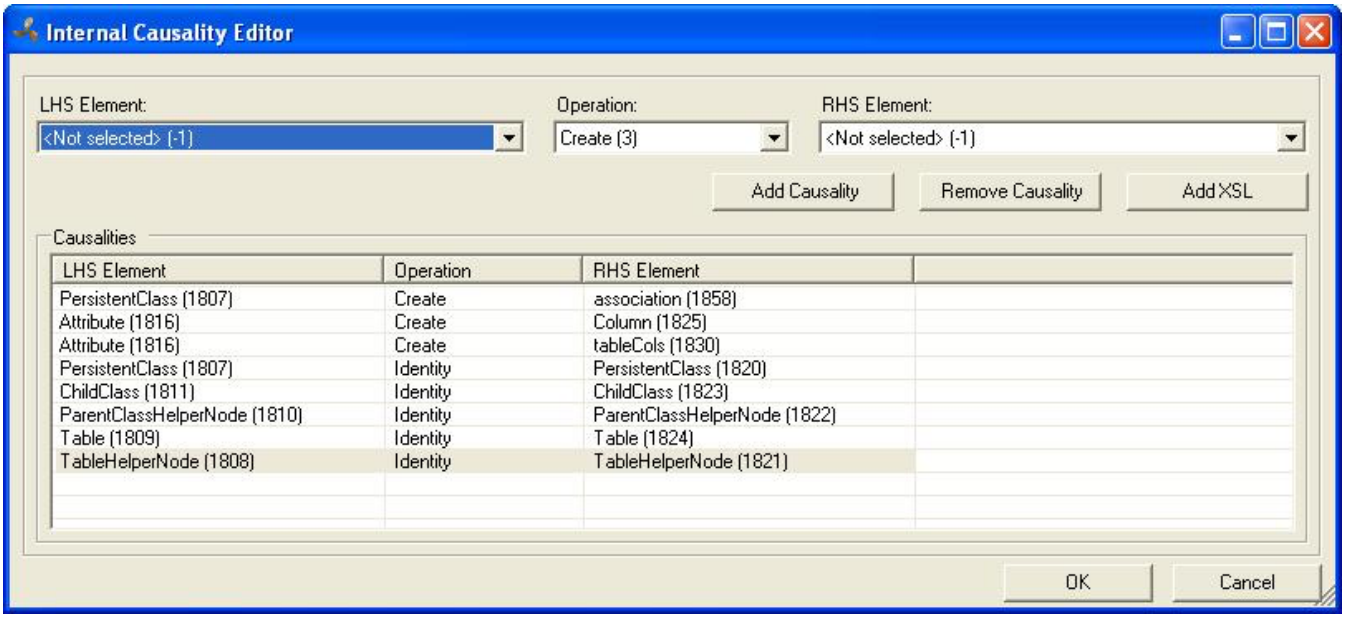


Rule *CreateParentClassHelper*

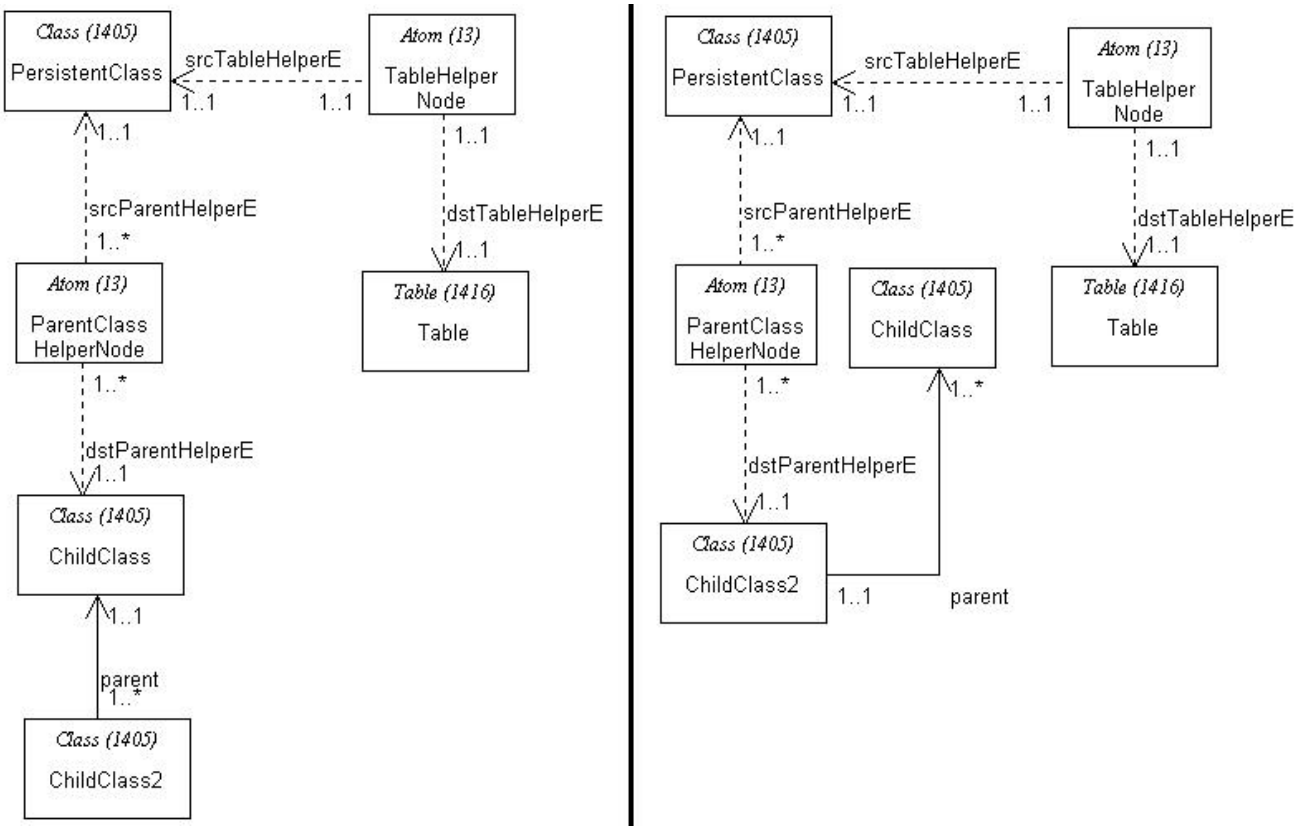


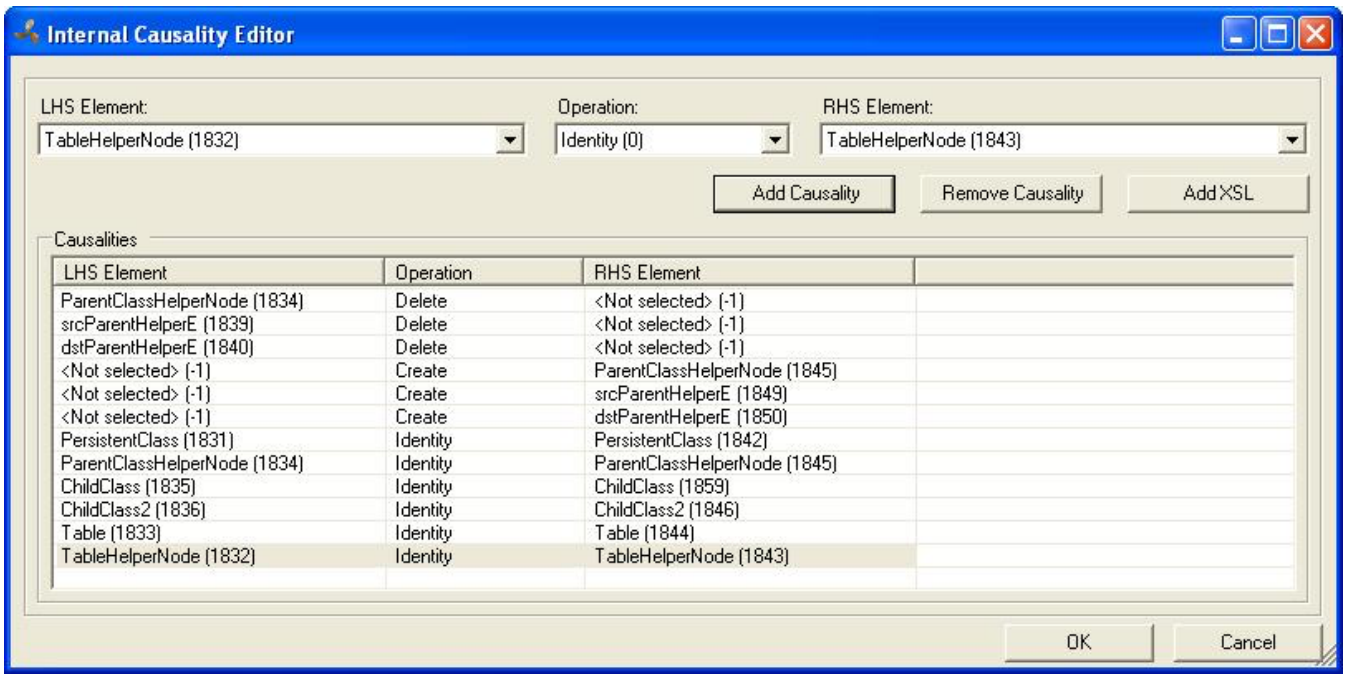
Rule *ProcessChildClass*



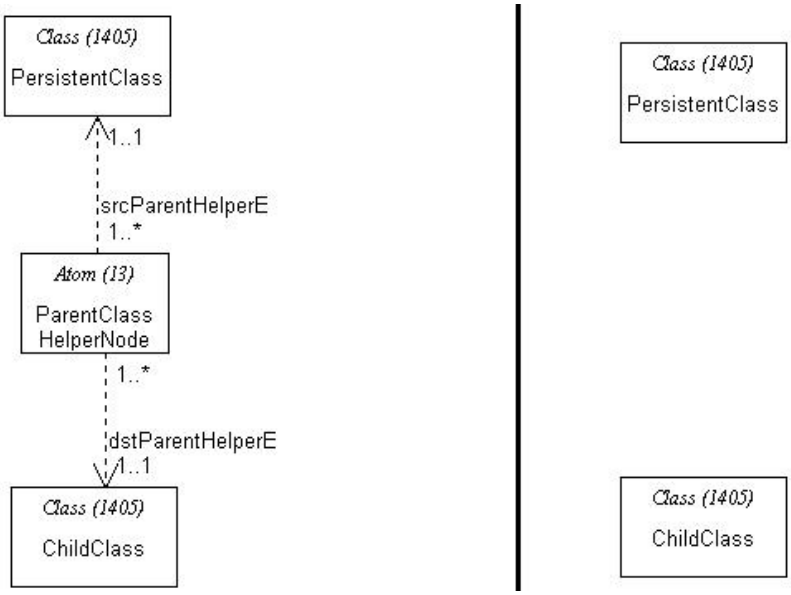


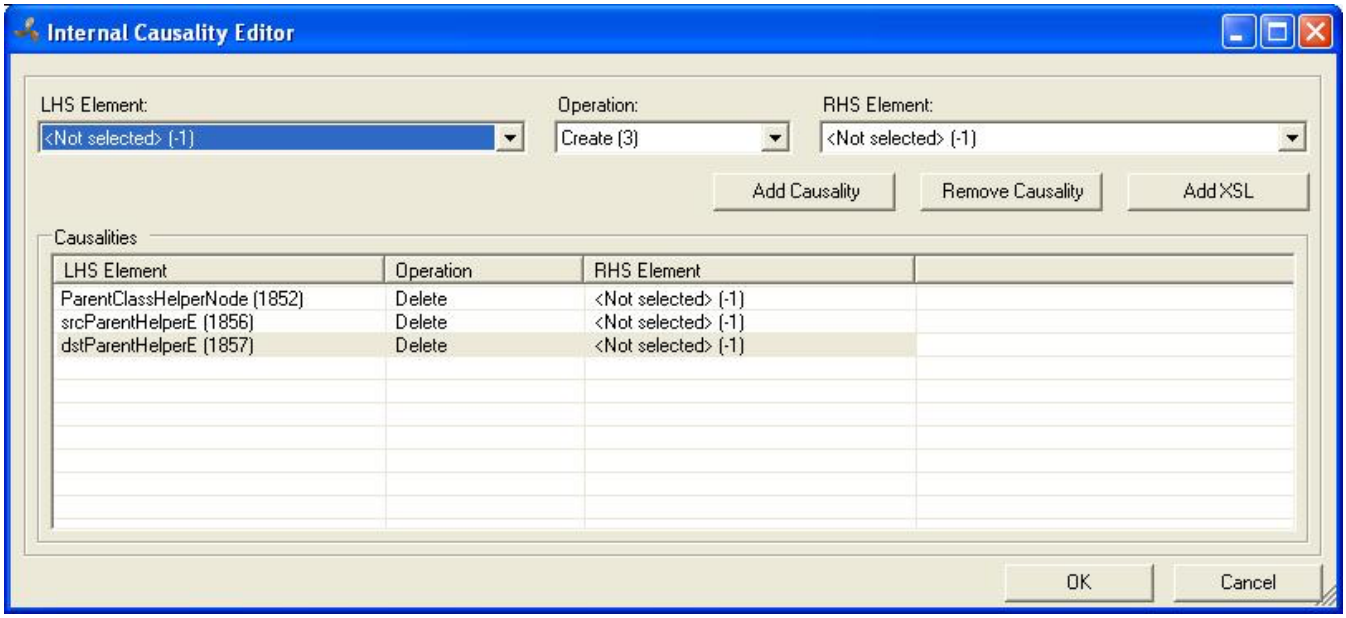
Rule *ShiftParentClassHelper*





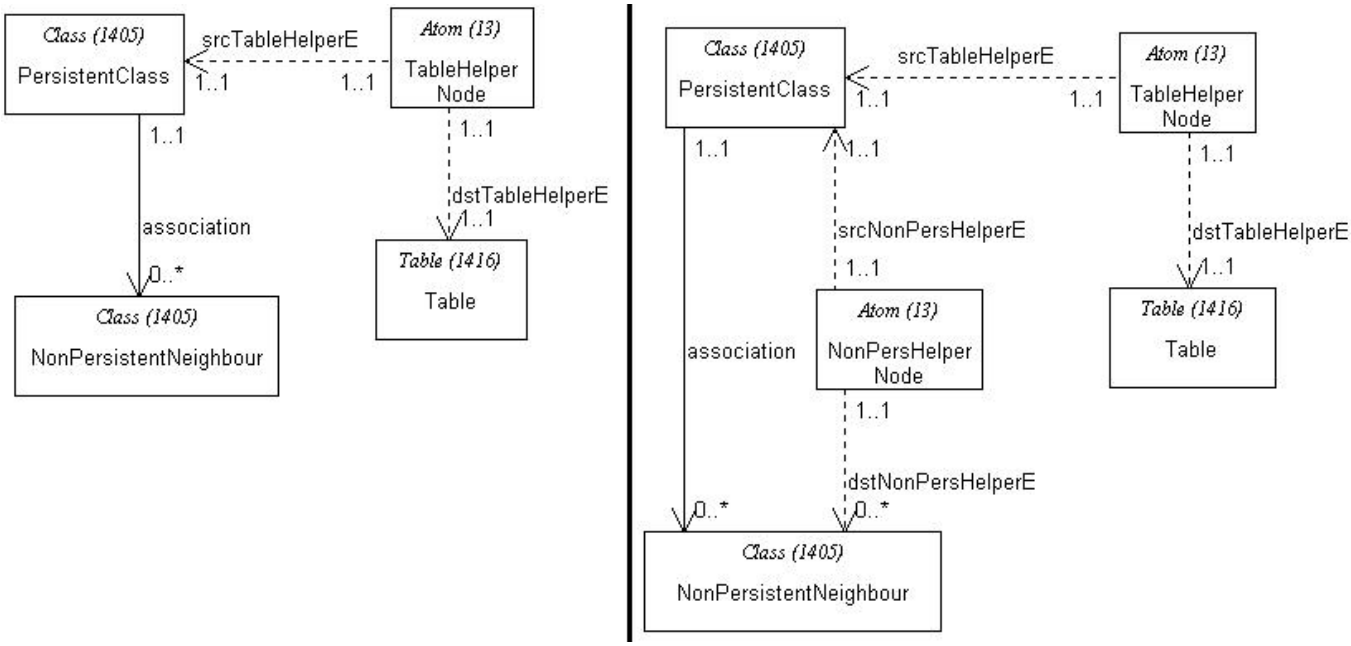
Rule *DeleteParentClassHelper*

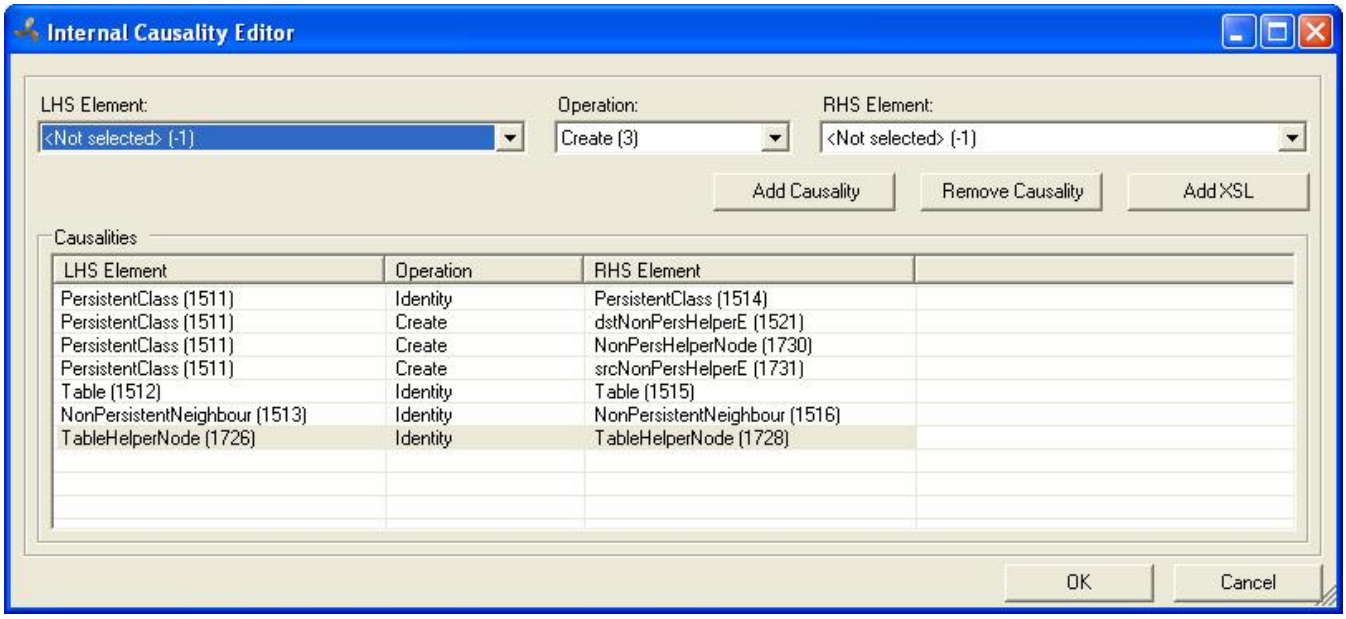




The rules (AddNonPersHelperEdge, ProcessNonPersClass, AddFkeyHelperEdge, ShiftNonPersHelperEdge, ShiftNonPersHelperEdge2, DeleteNonPersHelperEdge) are devoted to process the chains of non-persistent classes.

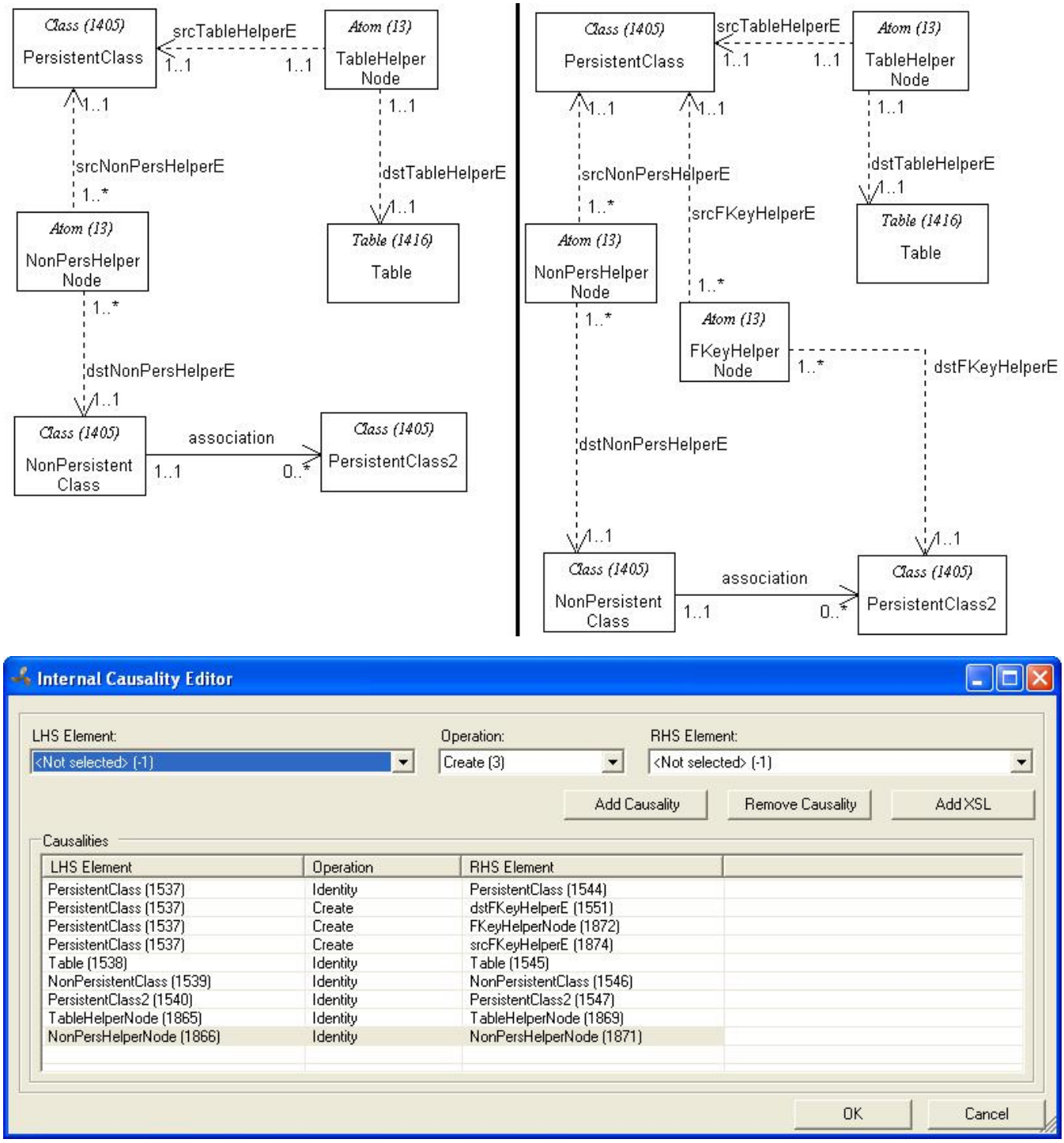
Rule *AddNonPersHelperEdge*
The *AddNonPersHelperEdge* rule matches a not processed persistent class (*PersistentClass*) with its created table (*Table*) along with its adjacent non-persistent classes (*NonPersistentClass*) and adds helper links (*NonPersHelperNode*) which connect them. The transformation uses these helper links to identify the next non-persistent classes. The non-persistent class chain could consist of optional number of classes and the transformation must to traverse and concatenate the names of the classes contained by the chain to create the table columns with these names. Therefore, the *NonPersHelperNodes* store the actual prefix of the table names created from the non-persistent class names. These helper links always point to the actual non-persistent classes being processed.



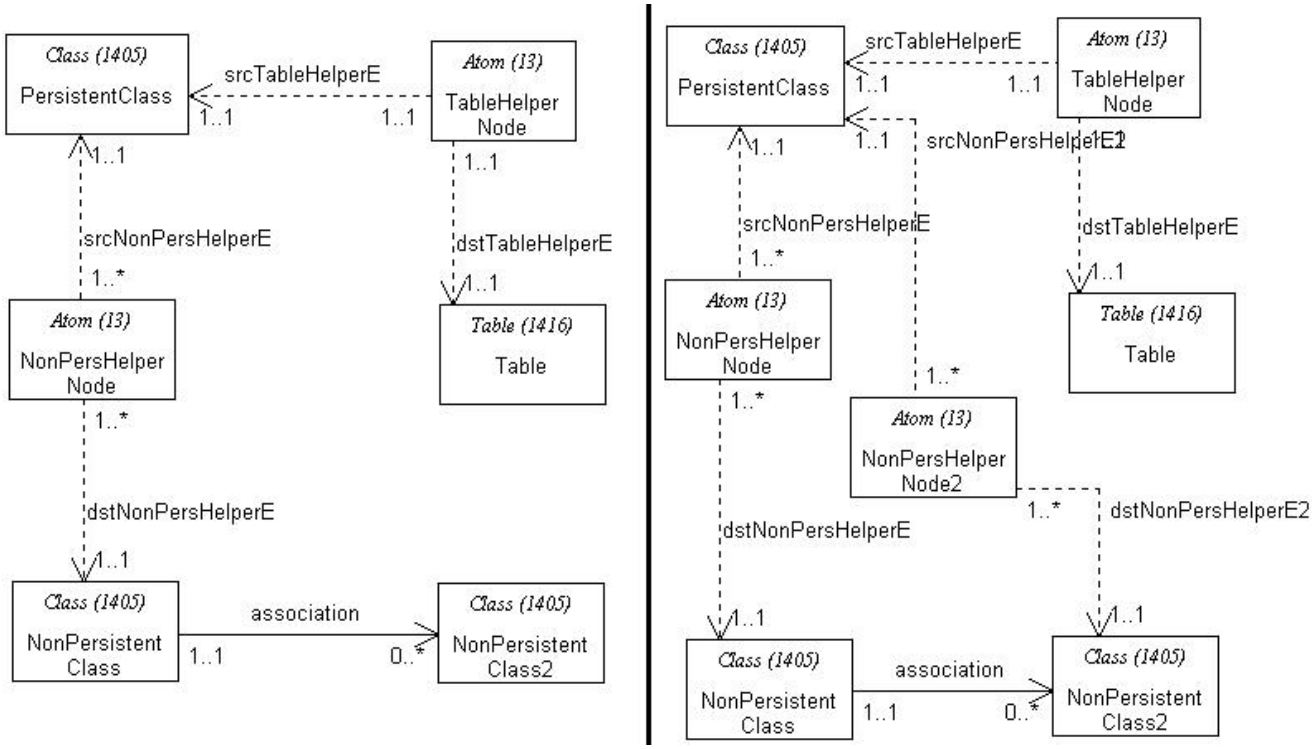


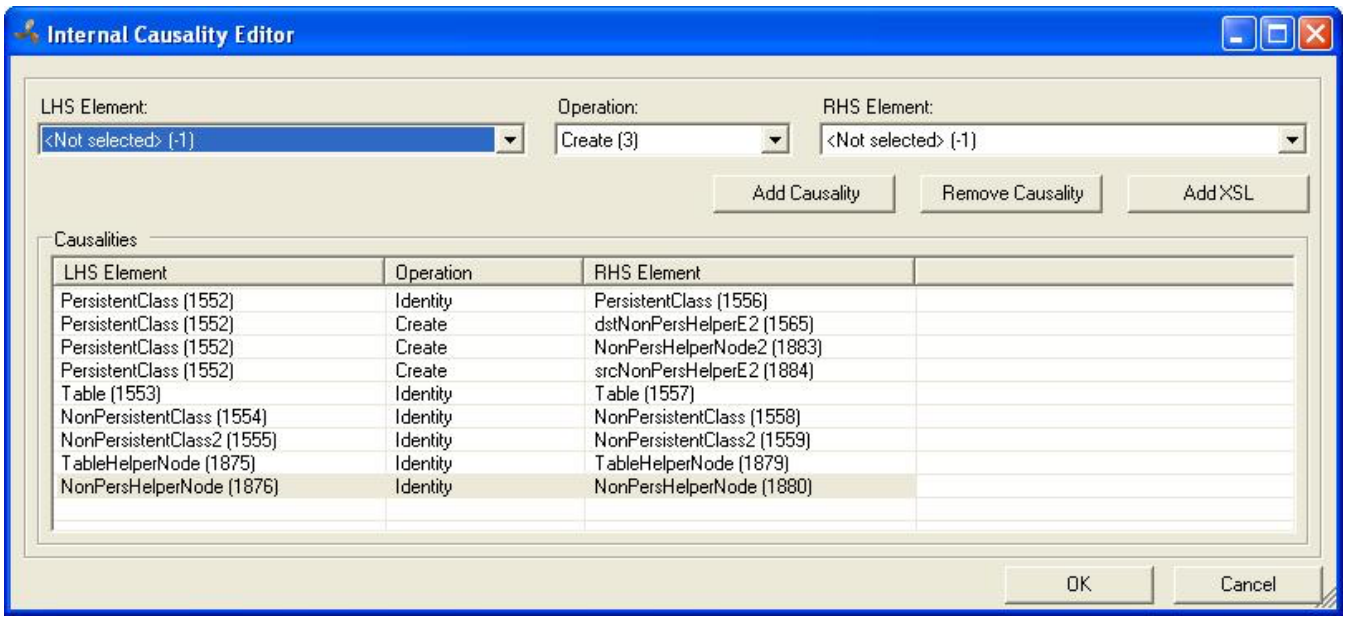
Rule *ProcessNonPersClass*
The *ProcessNonPersClass* uses the *NonPersHelperNodes* to transform the attributes of the pointed non-persistent classes to columns and public keys.

Rule *AddFKeyHelperEdge*
If the next class in the chain is a persistent class the *AddFKeyHelperEdge* rule links it with an *FKeyHelperNodes* to the actual persistent class.

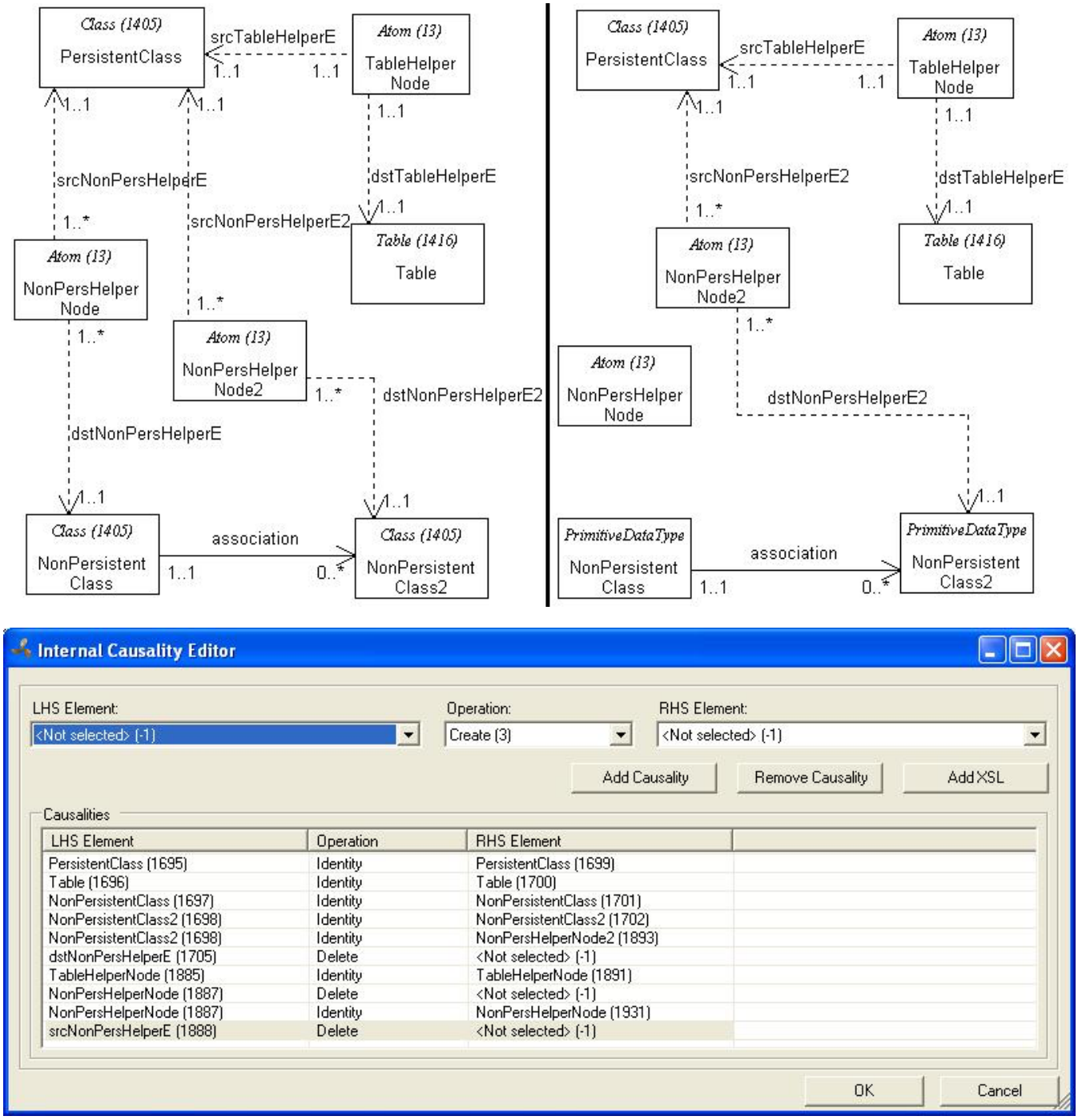


Rule *ShiftNonPersHelperEdge*
Otherwise, if the next class is also non-persistent, then the two *ShiftNonPersHelperEdge* rules shift the helper links from the actually pointed and already processed non-persistent classes to their adjacent non-persistent classes.

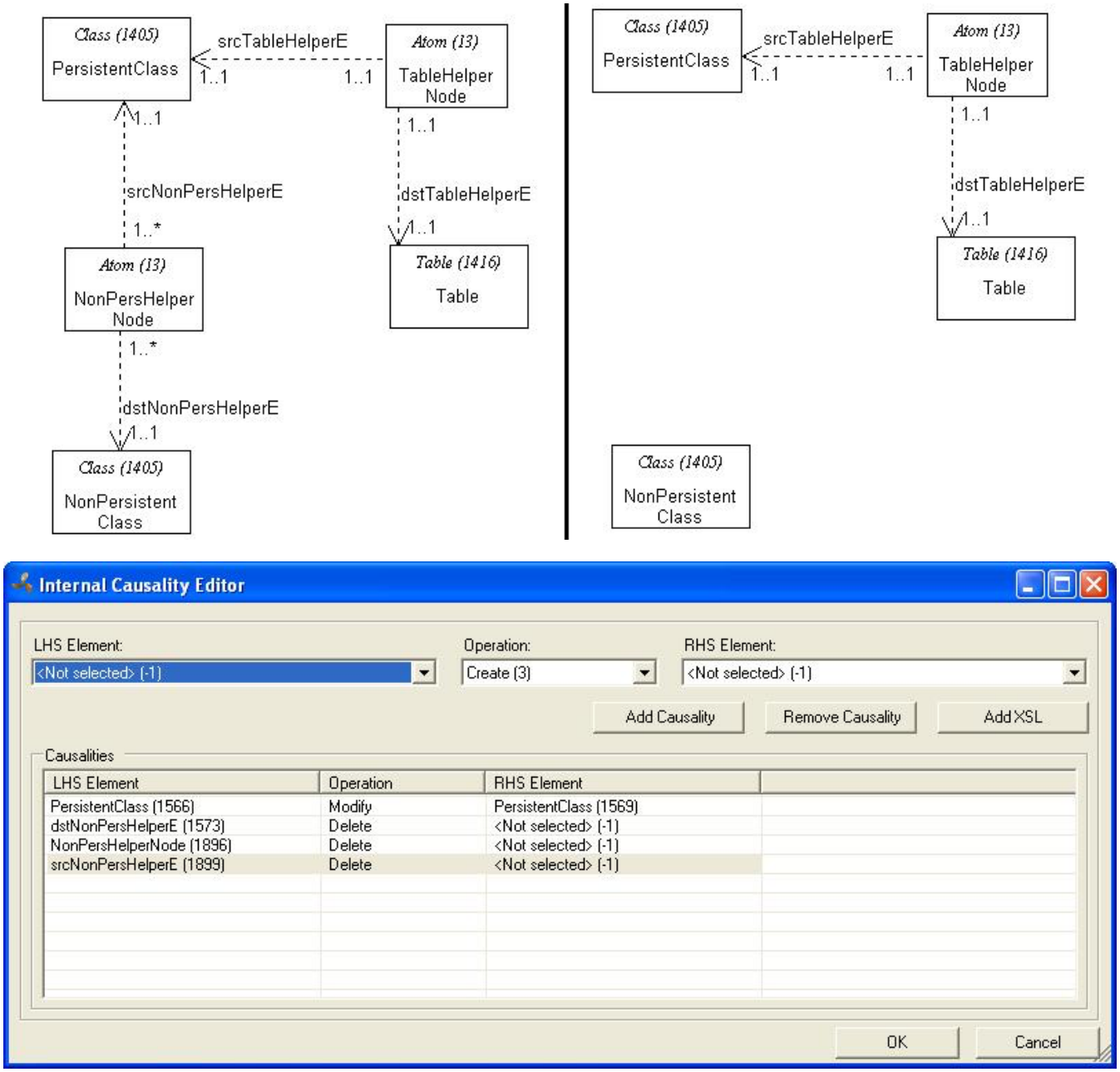




Rule *ShiftNonPersHelperEdge2*

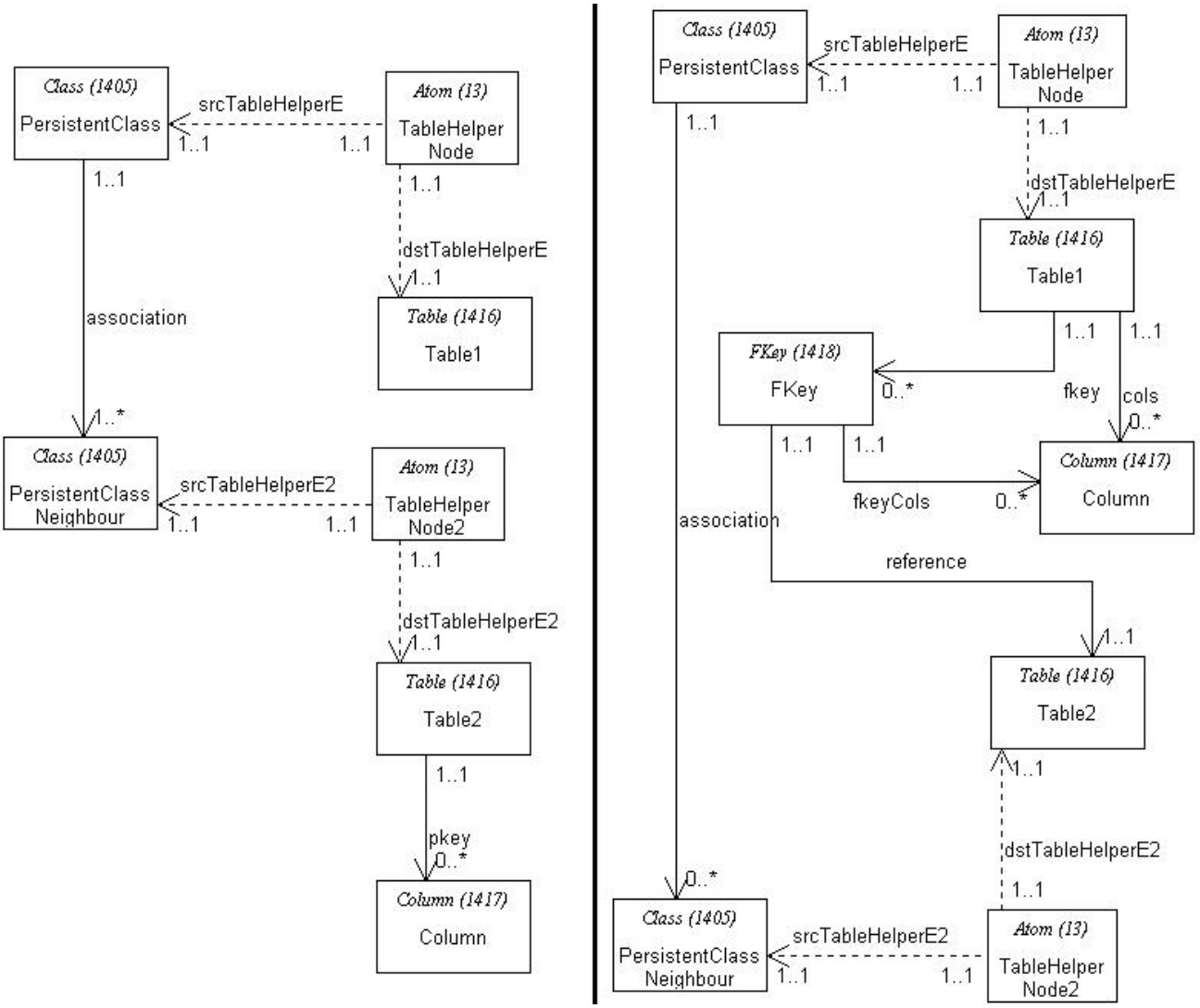


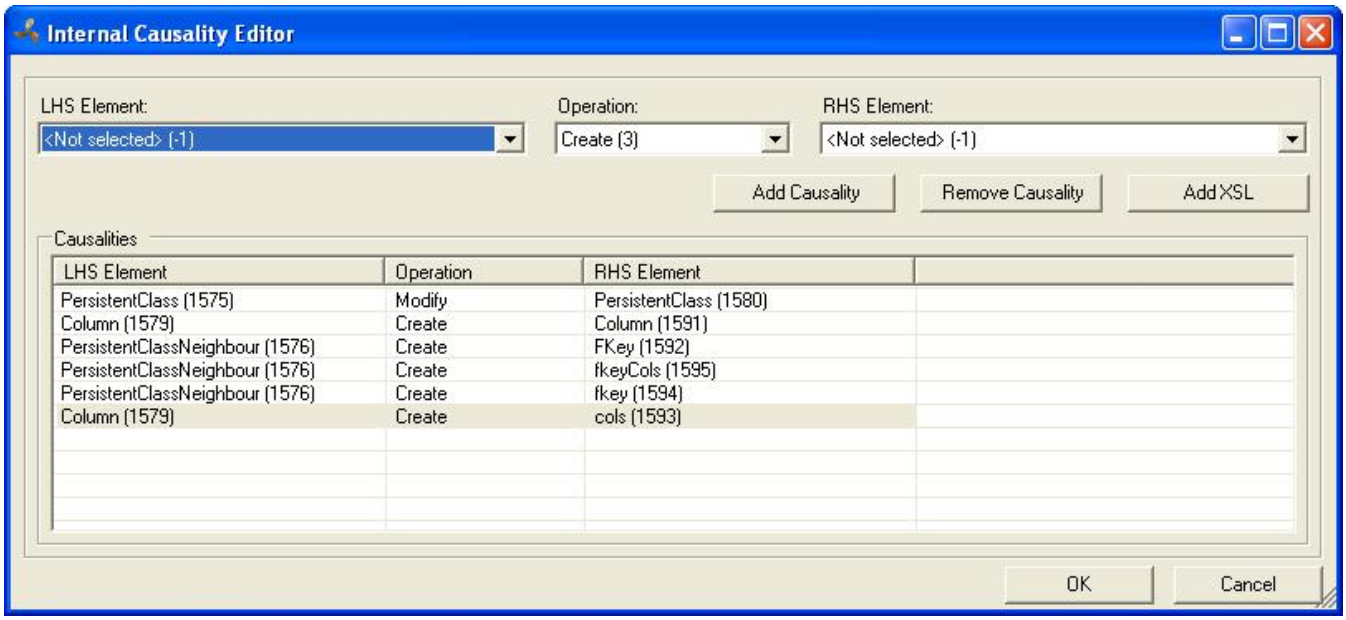
Rule *DeleteNonPersHelperEdge*
If all non-persistent classes in the current chain are processed, the branch to the *DeleteNonPersHelperEdge* rule is selected. This rule removes the *nonPersHelperEdges* and modifies the *is_processed* property of the actual persistent class to true (1).



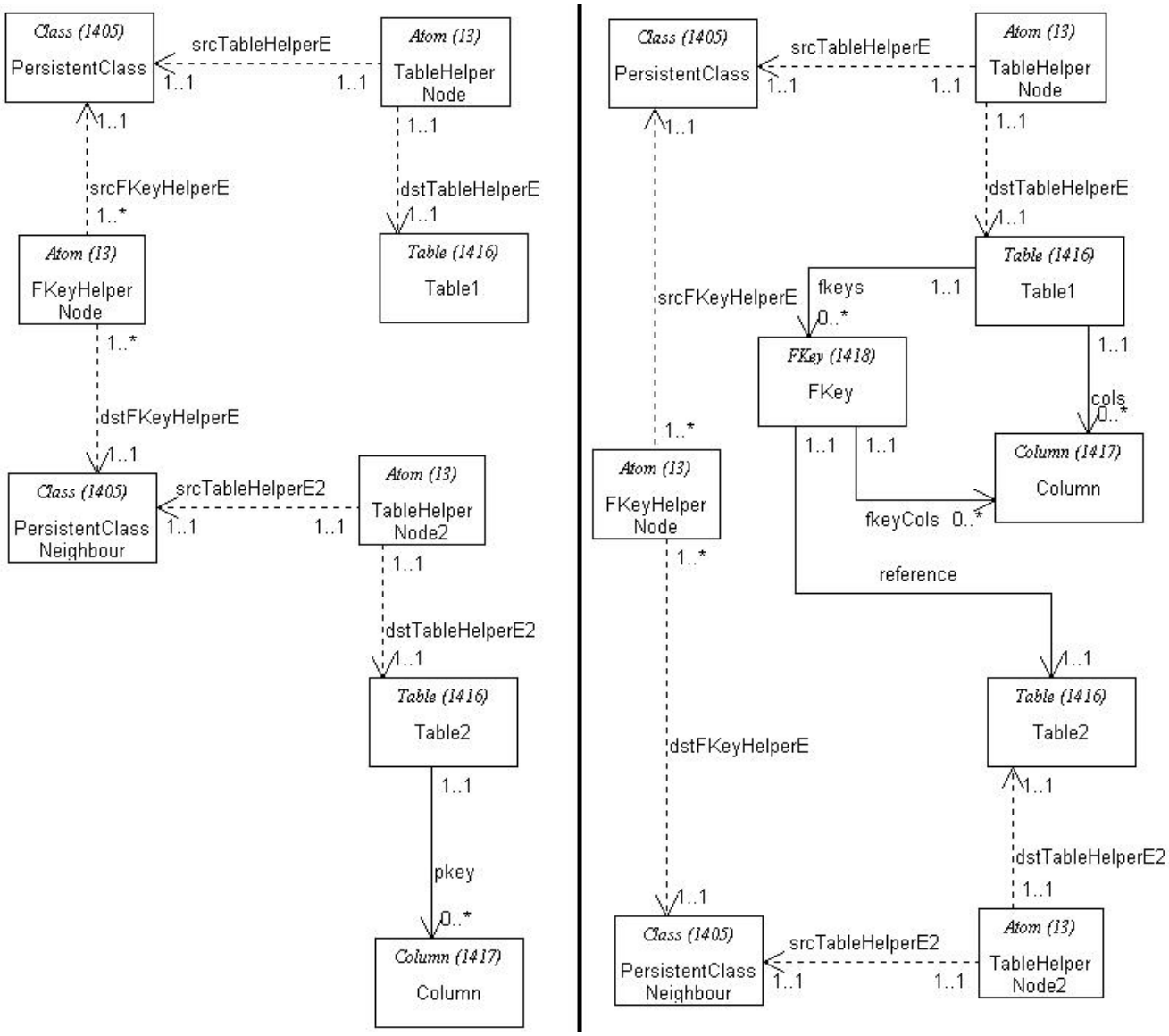
The *CreateFKeysFromNeighbours*, *CreateFKeysFromHelpers*, *DeleteFKeyHelperEdges*, *DeletTableHelperEdges* and *DeleteTemporaryAssociations* rules are executed exhaustively. The first two rules create the foreign key relations based on the primary keys of the adjacent persistent classes and the *fkeyHelperEdges* created by the *AddFKeyHelperEdge* rule. Finally, the last three rules remove the helper links.

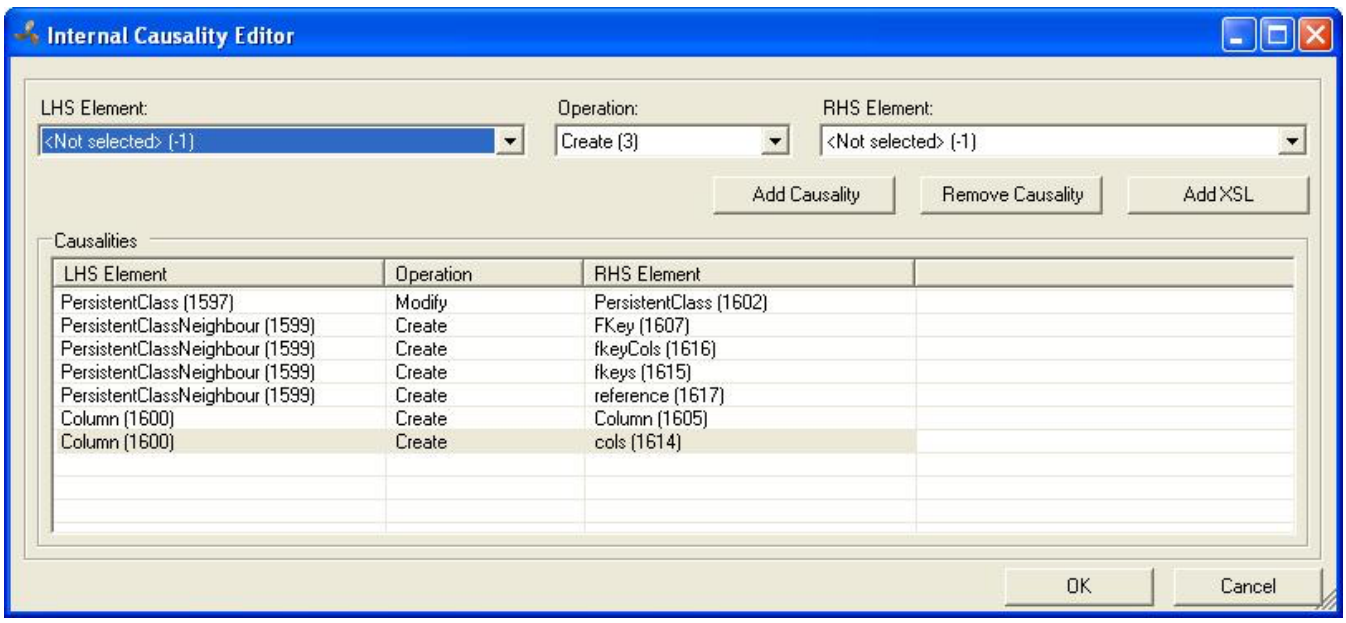
Rule *CreateFKeysFromNeighbours*



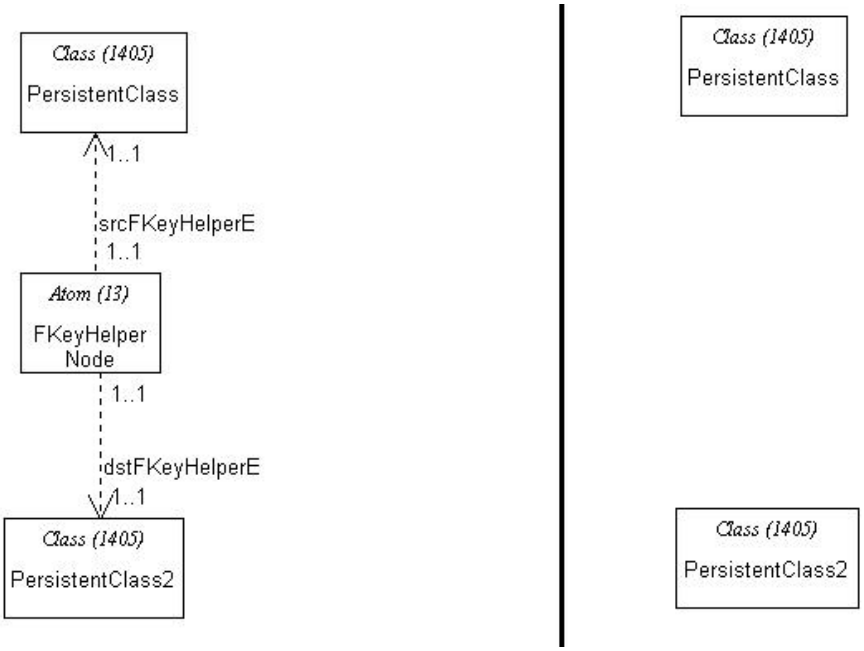


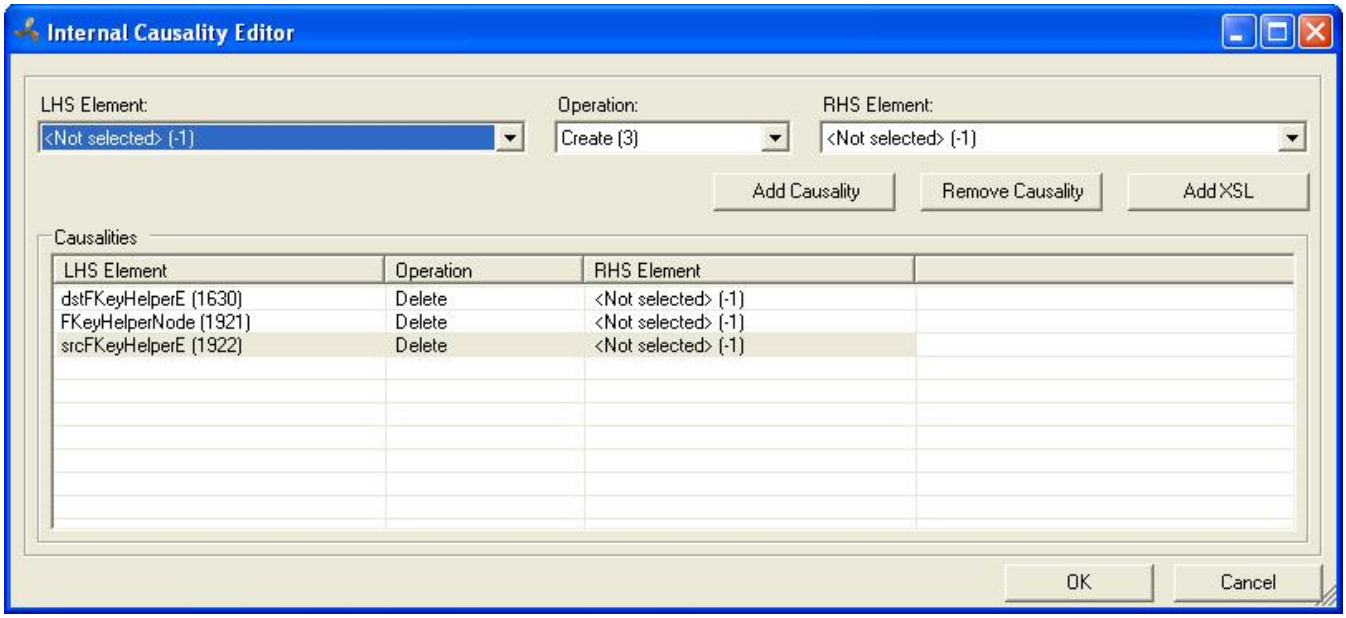
Rule *CreateFKeysFromHelpers*



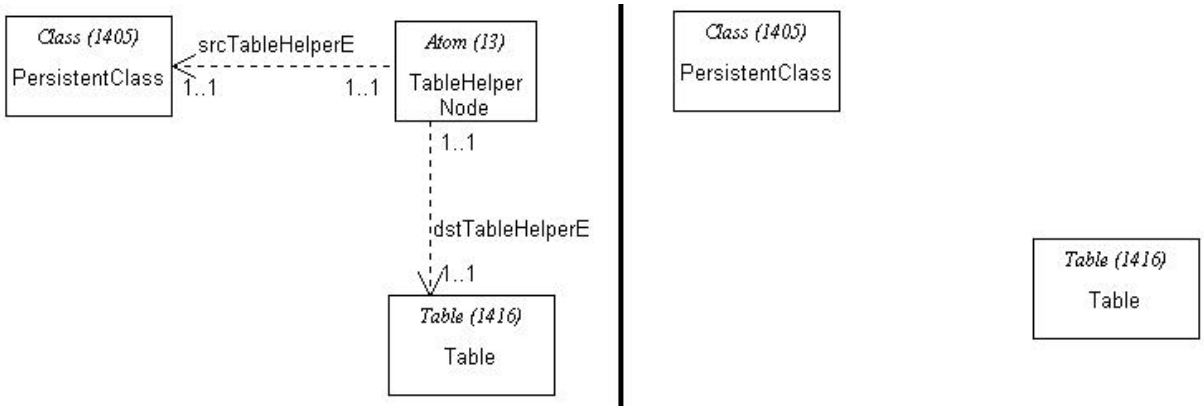


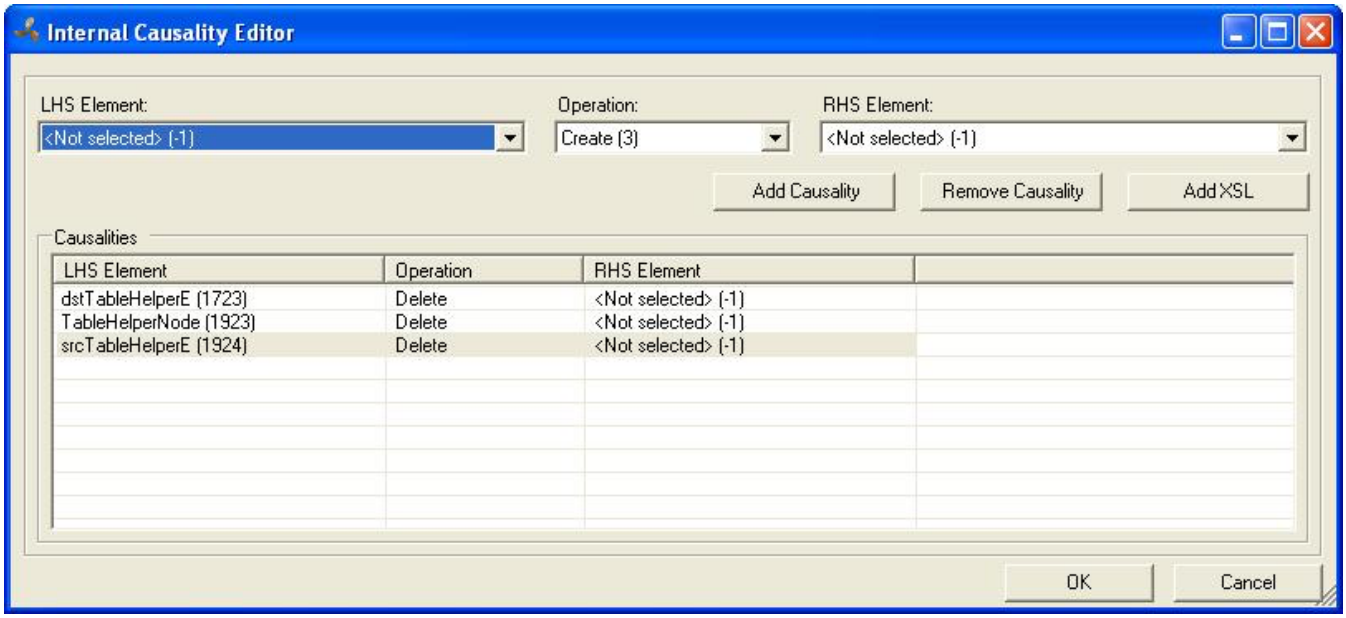
Rule *DeleteFKeyHelperEdge*



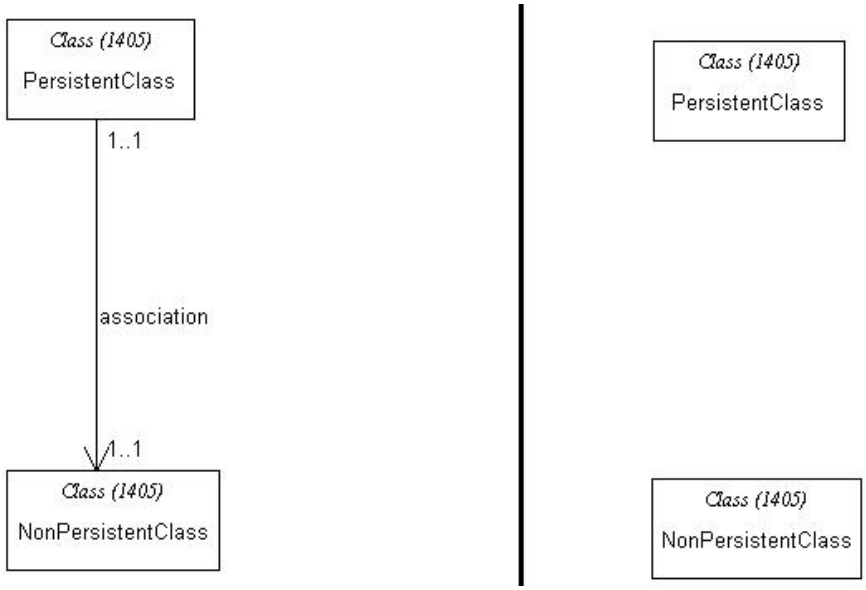


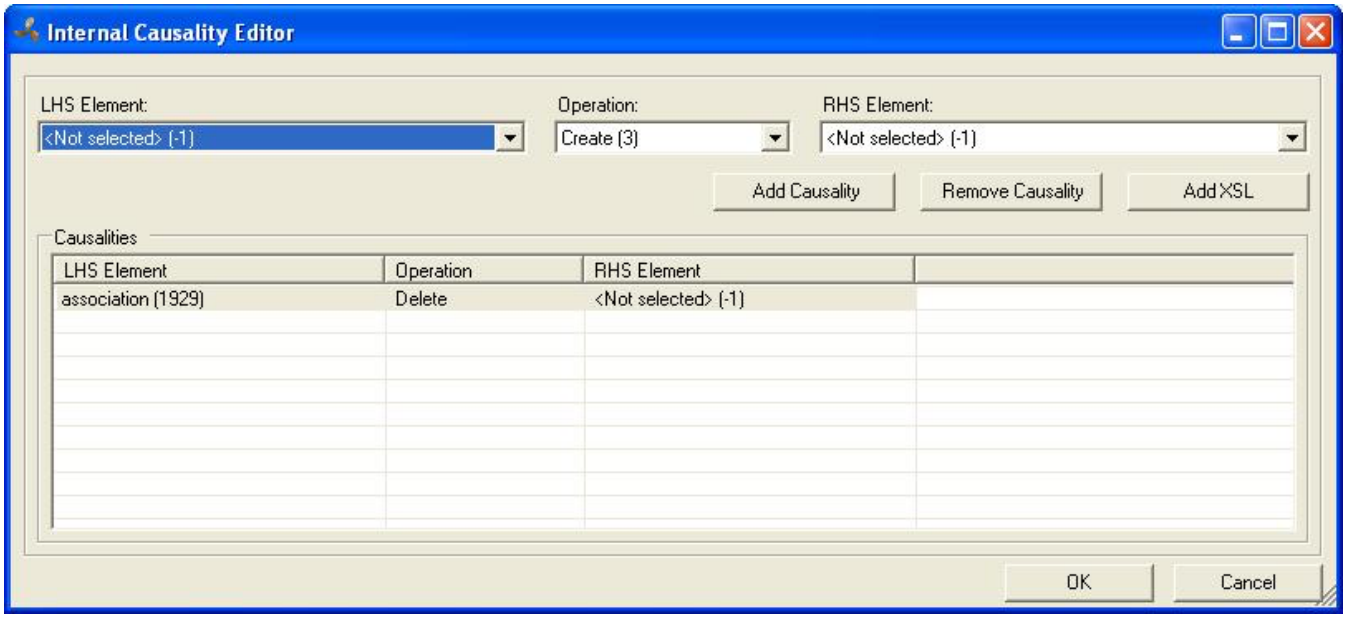
Rule *DeleteTableHelperEdge*





Rule *DeleteTemporaryAssociations*

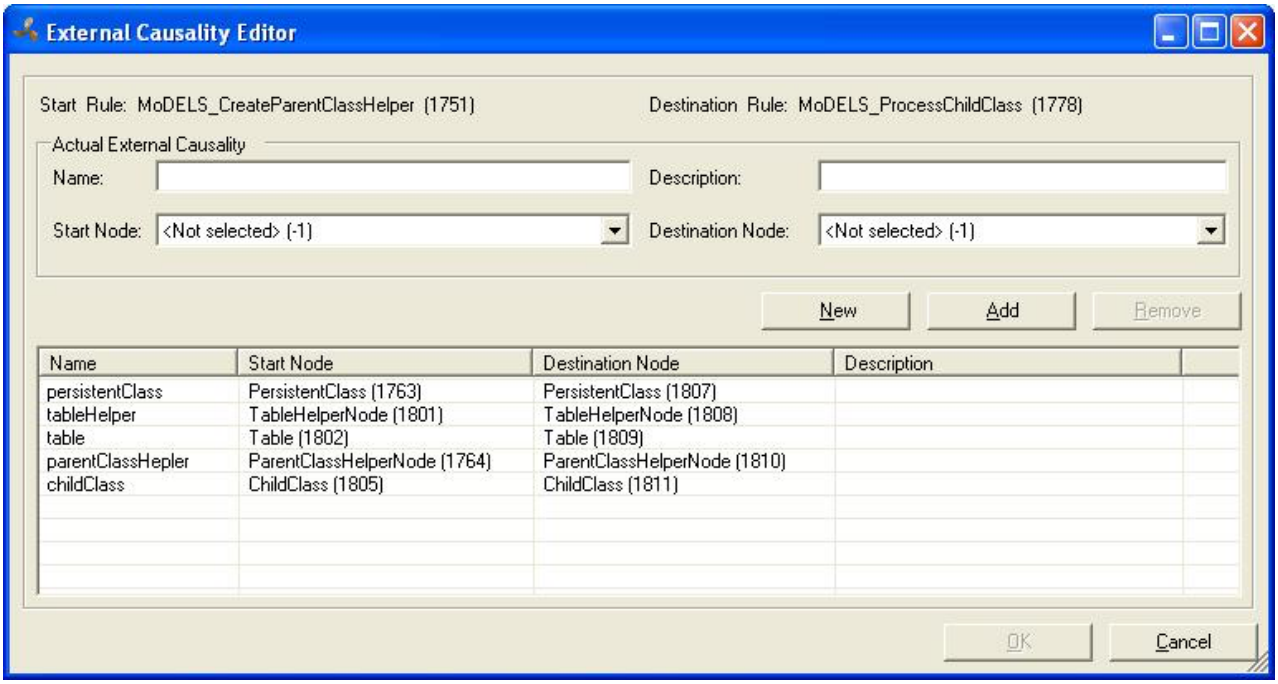




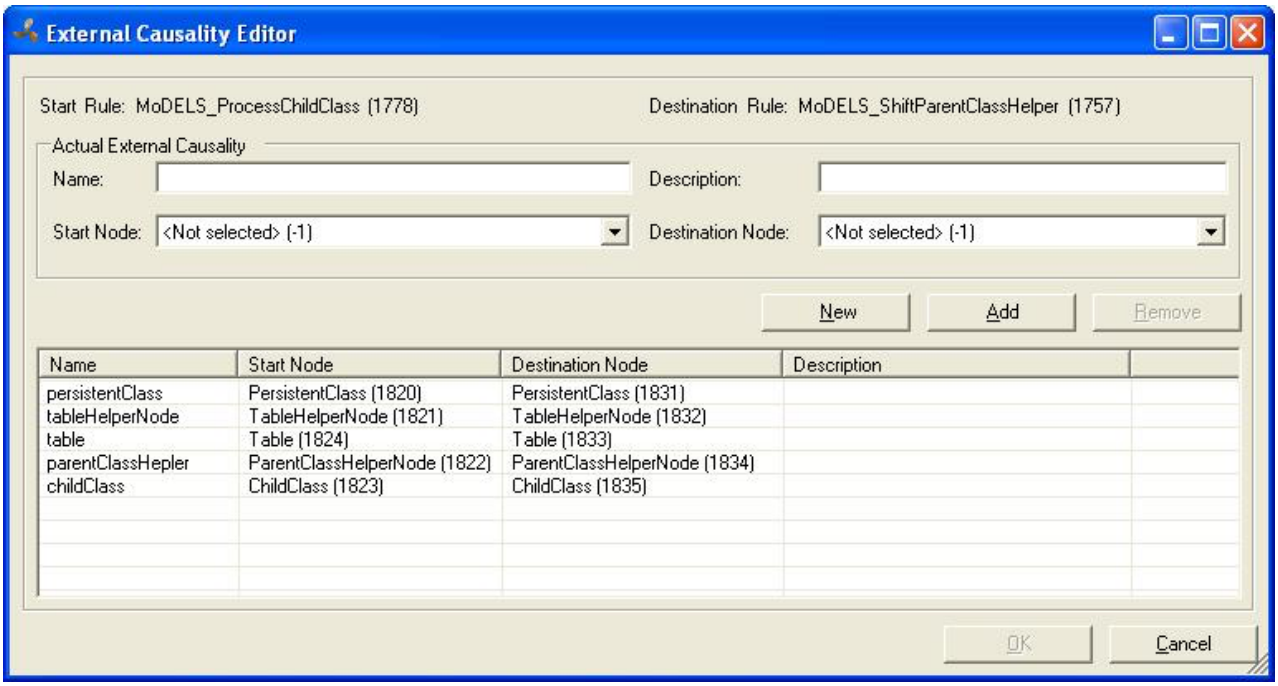
External Causalities

External causalities define the relations between the rules following each other, i.e. the right-hand side nodes and left-hand side nodes of the next rule.

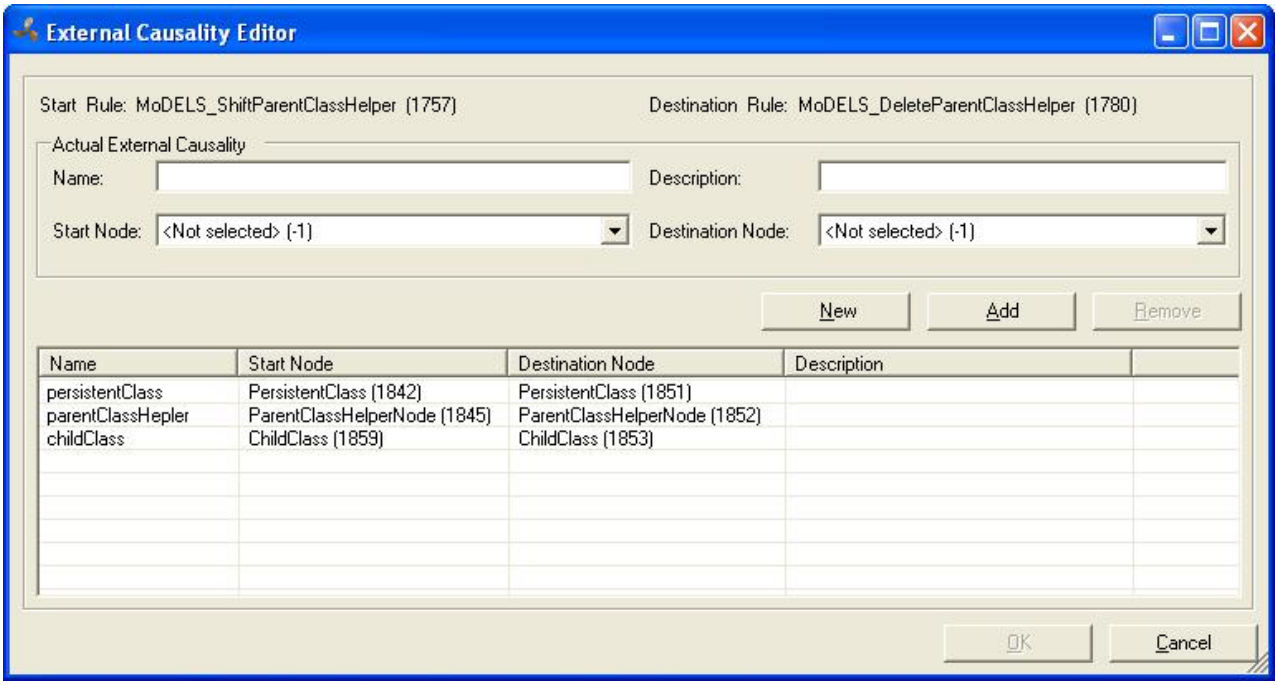
CreateParentClassHelper - ProcessChildClass



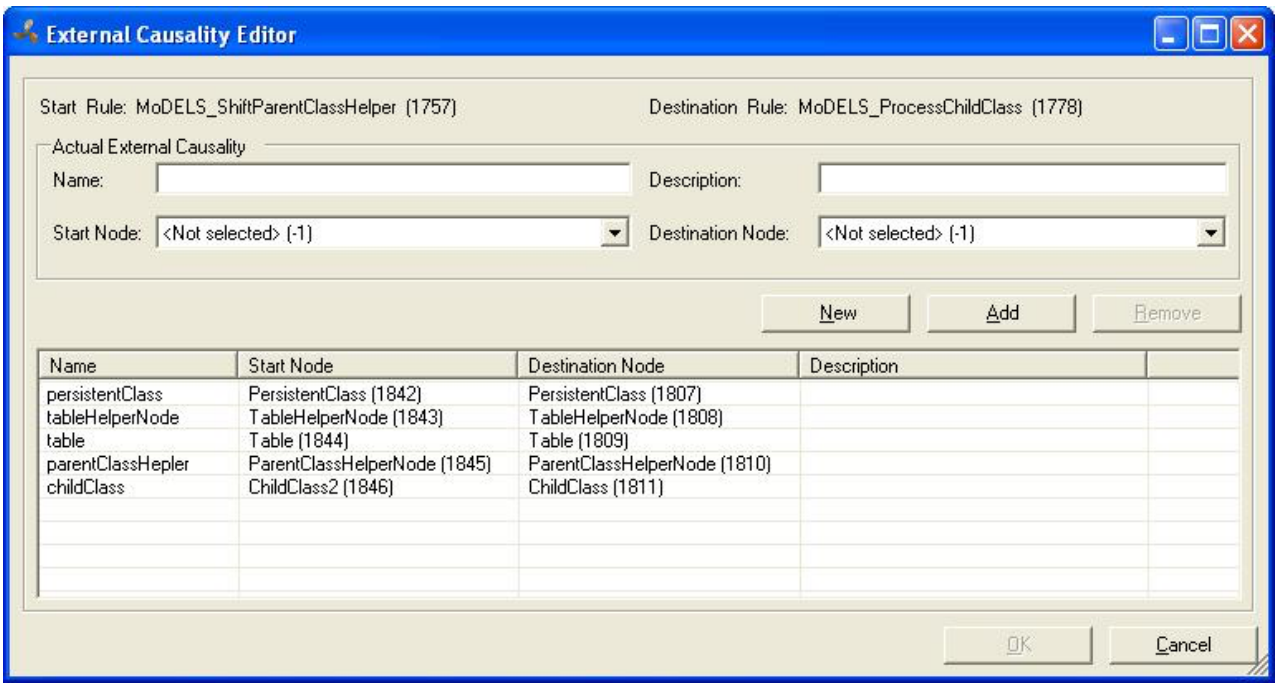
ProcessChildClass - ShiftParentClassHelper



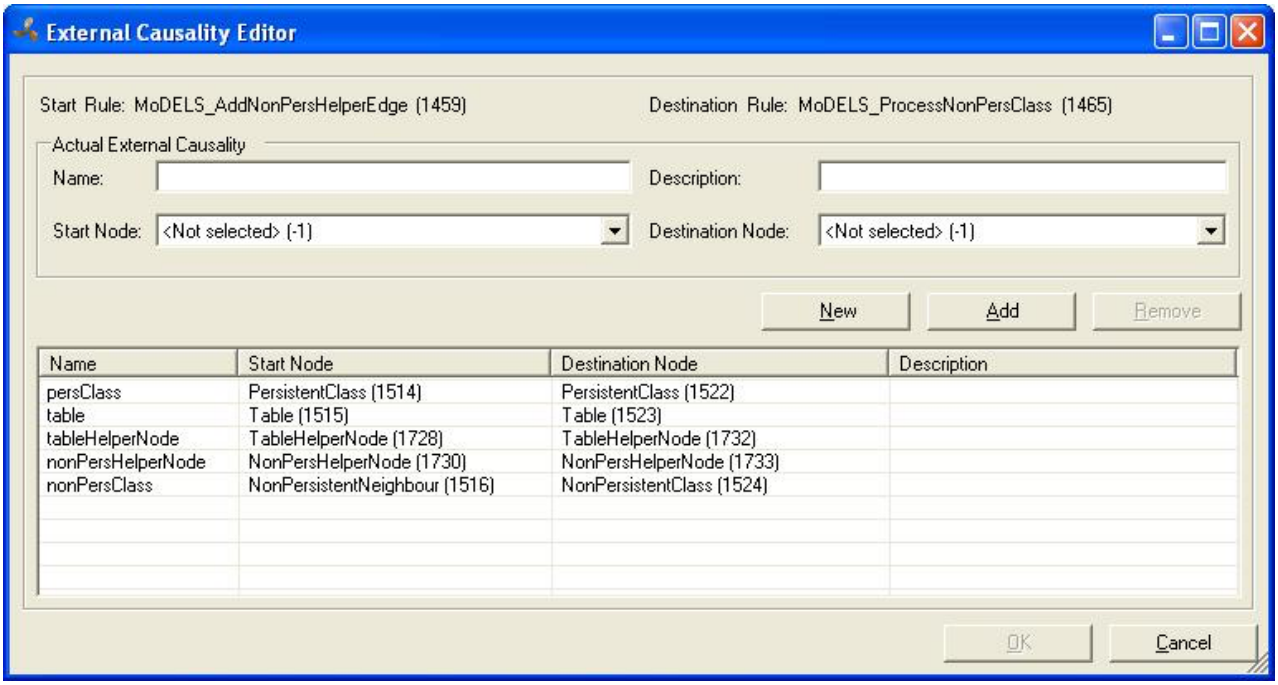
ShiftParentClassHelper - DeleteParentClassHelper



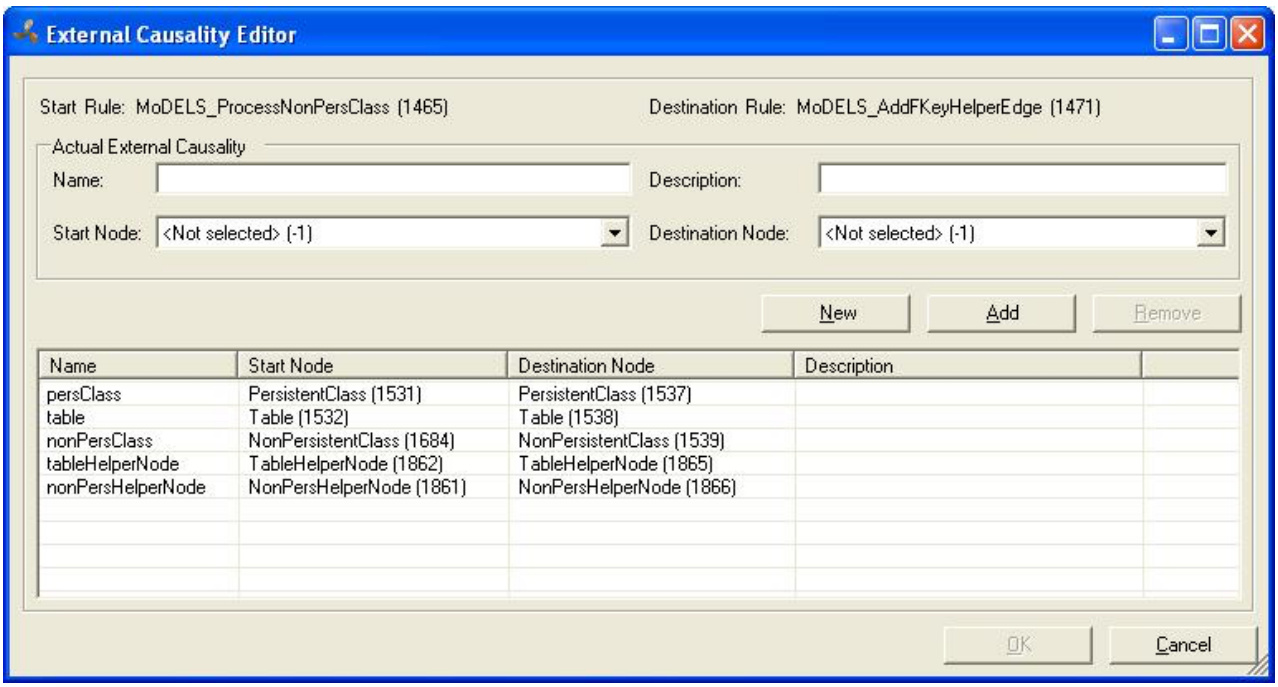
ShiftParentClassHelper - ProcessChildClass



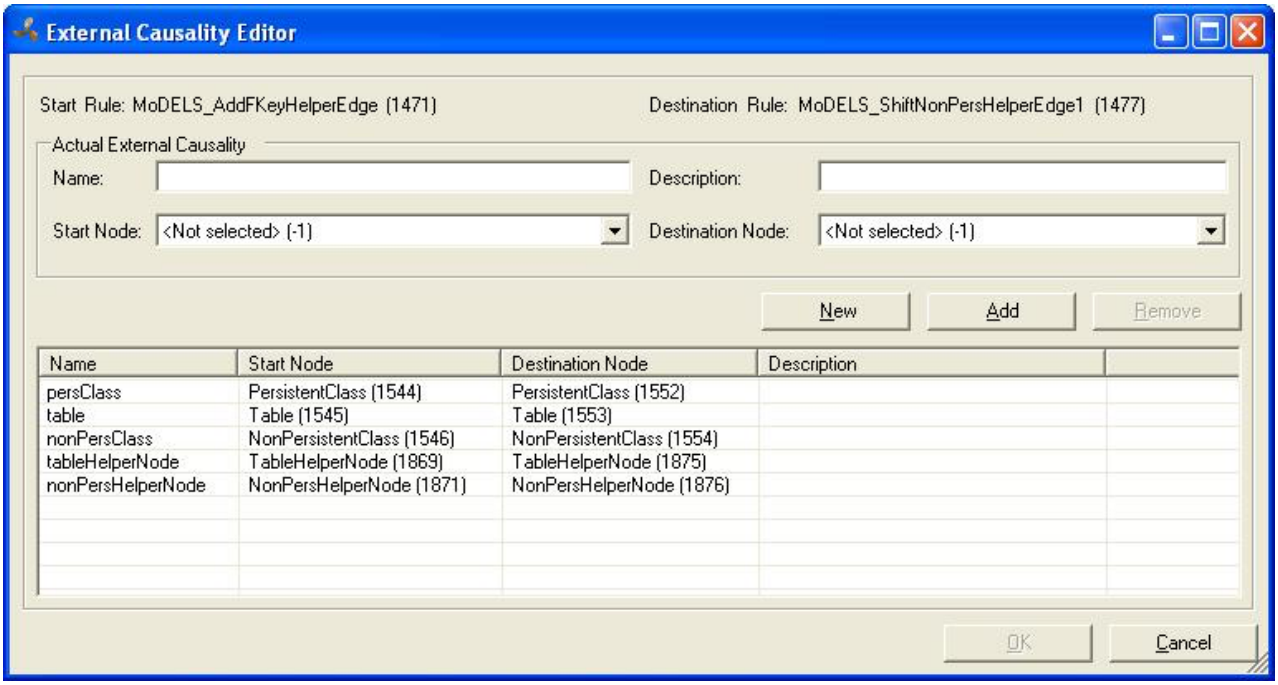
AddNonPersHelperEdge - ProcessNonPersClass



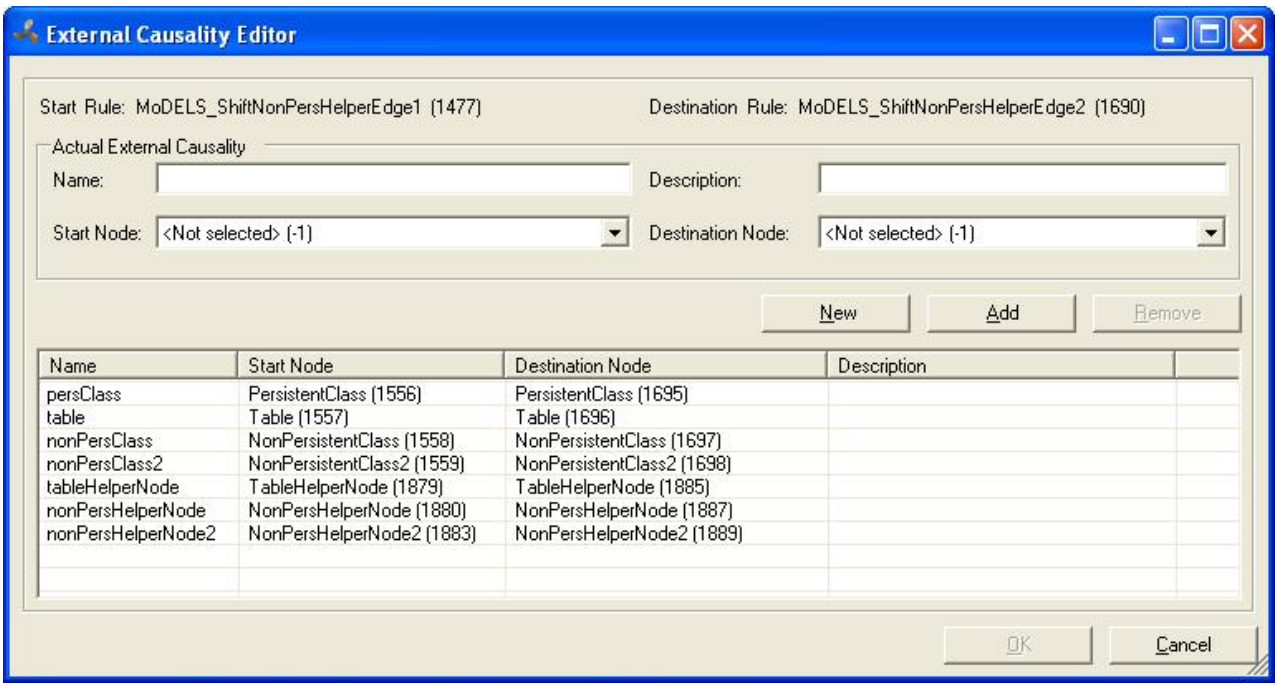
ProcessNonPersClass - AddFKeyHelperEdge



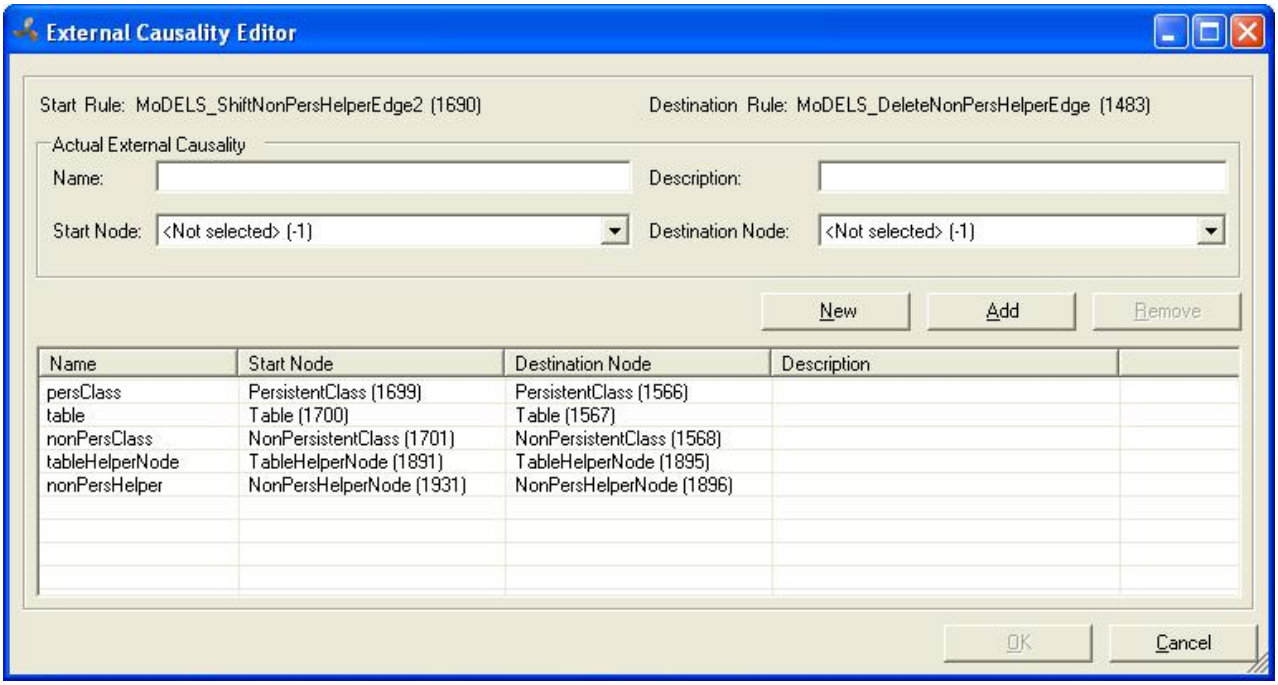
AddFKeyHelperEdge - ShiftNonPersHelperEdge



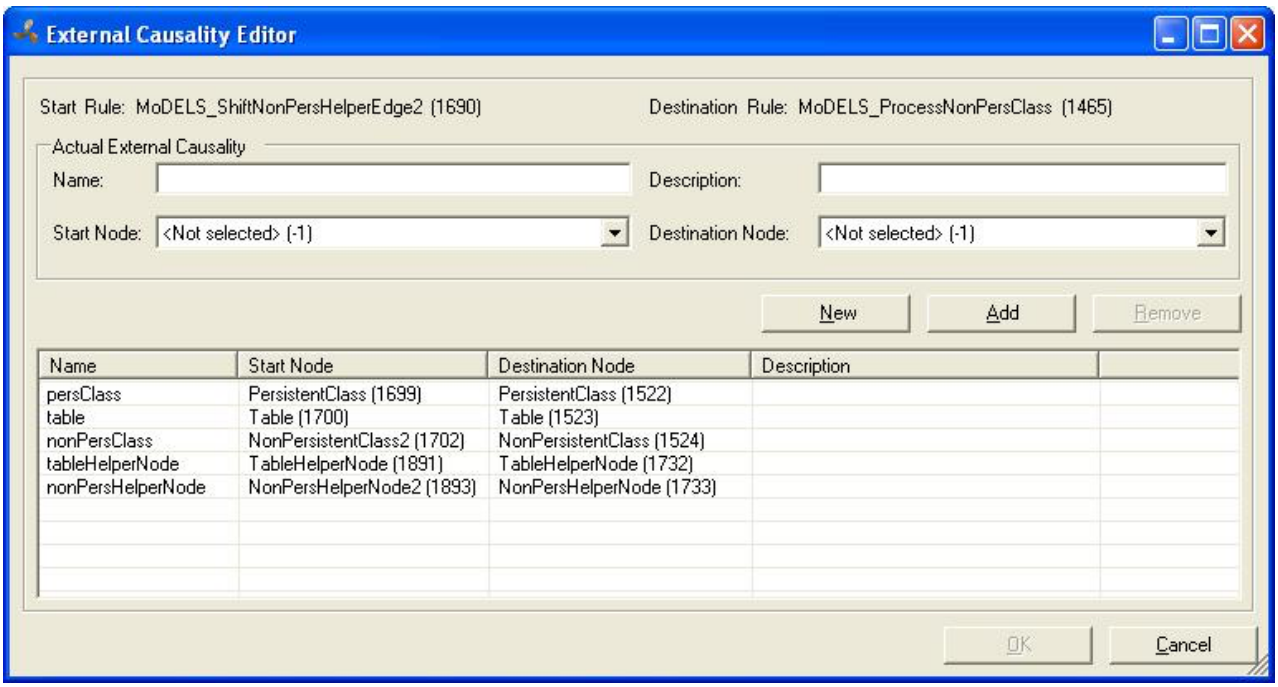
ShiftNonPersHelperEdge - ShiftNonPersHelperEdge2



ShiftNonPersHelperEdge2 - DeleteNonPersHelperEdge



ShiftNonPersHelperEdge2 - ProcessNonPersClass



XSL files of the transfromations

[Step CreateTable]
[Step AddNonPersHelperEdge]
[Step ProcessNonPersClass]
[Step AddFKeyHelperEdge]
[Step ShiftNonPersHelperEdge]
[Step CreateFKeysFromNeighbours]
[Step CreateFKeysFromHelpers]