به نام خدا

دانشگاه بو علی سینا

گزارش کار پیاده سازی

عنوان مقاله :مدل استدلال کننده ی خودکار برای قوانین وابستگی های تابعی با استفاده از شبکه های پتری رنگی

استاد راهنما : دکتر یونس سیفی

نویسنده : محمد پیشدار (کارشناسی ارشد فناوری اطلاعات)

مقدمه :

مدیران پایگاه داده نیاز به محاسبه استنتاج های از وابستگی های تابعی برای نرمال سازی پایگاه های داده دارند . شبکه های پتری رنگی ابزاری قدرتمند برای بررسی سیستم های مختلف می باشد . در این مقاله با استفاده از قوانین آرمسترانگ استدلال خودکار یک وابستگی تابعی از وابستگی های تابعی اولیه انجام شده است .

در طراحی پایگاه های داده ی ارتباطی یکی از گام های اساسی نرمال سازی حذف آنومالی ها می باشد . نرمال سازی پایگاه داده نیاز به استخراج وابستگی های تابعی بین صفت ها ی پایگاه داده دارد و با استفاده از قوانین آرمسترانگ برای نرمال سازی نیاز به نتیجه گیری وابستگی های تابعی جدید از وابستگی های تابعی اولیه می باشد .

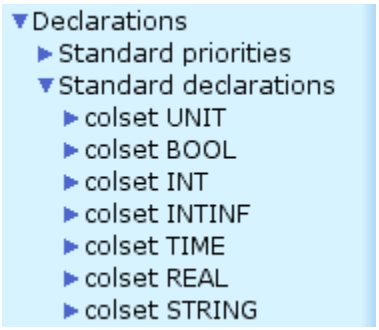
قوانین آرمسترانگ :

```
Augmentation: if A →B, then AC →BC
Transitivity: if A →B and B →C, then A →C
Reflexivity: if B is a subset of A, then A →B
Self-determination: A→A
Decomposition: if A →BC, then A →B and A →C
Union: if A →B and A →C, then A →BC
Composition: if A →B and C →D, then AC →BD
General Unification: if A →B and C →D, then A∪(C-B)→BD
```

شروع پیاده سازی :

تعریف ColorSet **:**

ابتدا ColSet **های مر بوط به مدل را در نرم افزار** Cpn Tools **و در منوی سمت چپ و در قسمت** Declarations **به صورت زیر تعریف می کنیم . (زبان پیاده سازی** ML **می باشد)**



قسمت Declarations

```
colset ATTRIBUTE = with A | B | C | D| E | F;
colset PRODRULE = with IN|SE|AU|GE|CO|UN|DE|TR;
colset PREDRULELIST = list INT;
```

```
colset ATTRIBUTELIST = list ATTRIBUTE;
colset RULEGENERATION = product PRODRULE *PREDRULELIST;
colset FD = record N:INT * F:ATTRIBUTELIST * S: ATTRIBUTELIST *
G:RULEGENERATION;
colset RULES = list FD;
colset TOKEN = with t;
```

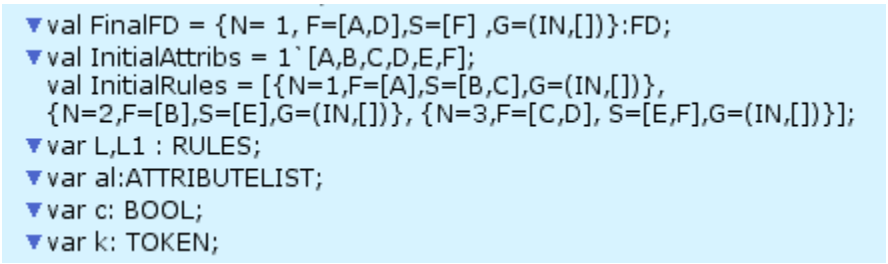
استفاده شده برای نمایش صفت های پایگاه داده	Colset Attribute
برای تعریف قوانین آرمسترانگ استفاده شده است (خلاصه ی اول عبارت ها).	colset PRODRULE
لیست شماره ی قوانینی که برای نتیجه گیری وابستگی تابعی جدید با استفاده از قوانین آرمسترانگ به کار رفته است .	colset PREDRULELIST
شامل لیستی از صفت های پایگاه داده می باشد .	colset ATTRIBUTELIST
شامل قانون آرمسترانگ و لیست صفت ها .	colset RULEGENERATION
نشان دهنده ی یک وابستگی تابعی می باشد . که شامل شماره و صفت های ابتدای وابستگی تابعی ، صفت های انتهایی وابستگی تابعی و قوانینی که با استفاده از آن این وابستگی تابعی نتیجه گیری شده است .	colset FD
برای ایجاد زمانبندی و ترتیب .	closet Token

```
▶ colset REAL
▼ colset STRING = string;
▼ colset ATTRIBUTE = with A | B | C | D| E | F;
▼ colset PRODRULE = with IN|SE|AU|GE|CO|UN|DE|TR;
▼ colset PREDRULELIST = list INT;
▼ colset ATTRIBUTELIST = list ATTRIBUTE;
▼ colset RULEGENERATION = product PRODRULE *PREDRULELIST;
▼ colset FD = record N:INT * F:ATTRIBUTELIST * S: ATTRIBUTELIST *
  G:RULEGENERATION;
▼ colset RULES = list FD;
▼ colset TOKEN = with t;
▼ val FinalFD = {N= 1, F=[A,D],S=[F] ,G=(IN,[])}:FD;
▼ val InitialAttribs = 1`[A,B,C,D,E,F];
  val InitialRules = [{N=1,F=[A],S=[B,C],G=(IN,[])},
  {N=2,F=[B],S=[E],G=(IN,[])}, {N=3,F=[C,D], S=[E,F],G=(IN,[])}];
▶ var L L1
▶ var al
▶ var c
▼ var k: TOKEN;
```

تعریف Colset **ها در** Cpn tools

تعریف مارکینگ اولیه و متغییر ها :

```
val InitialAttribs = 1`[A,B,C,D,E,F];
val InitialRules = [{N=1,F=[A],S=[B,C],G=(IN,[])},
{N=2,F=[B],S=[E],G=(IN,[])}, {N=3,F=[C,D], S=[E,F],G=(IN,[])}];
val FinalFD = {N= 1, F=[A,D],S=[F] ,G=(IN,[])}:FD;

1: A →BC
2: B →E
3: CD →EF

var L,L1 : RULES; var al:ATTRIBUTELIST;
var c: BOOL; var k: TOKEN;
```



مارکینگ اولیه و متغییر ها در Cpn Tools

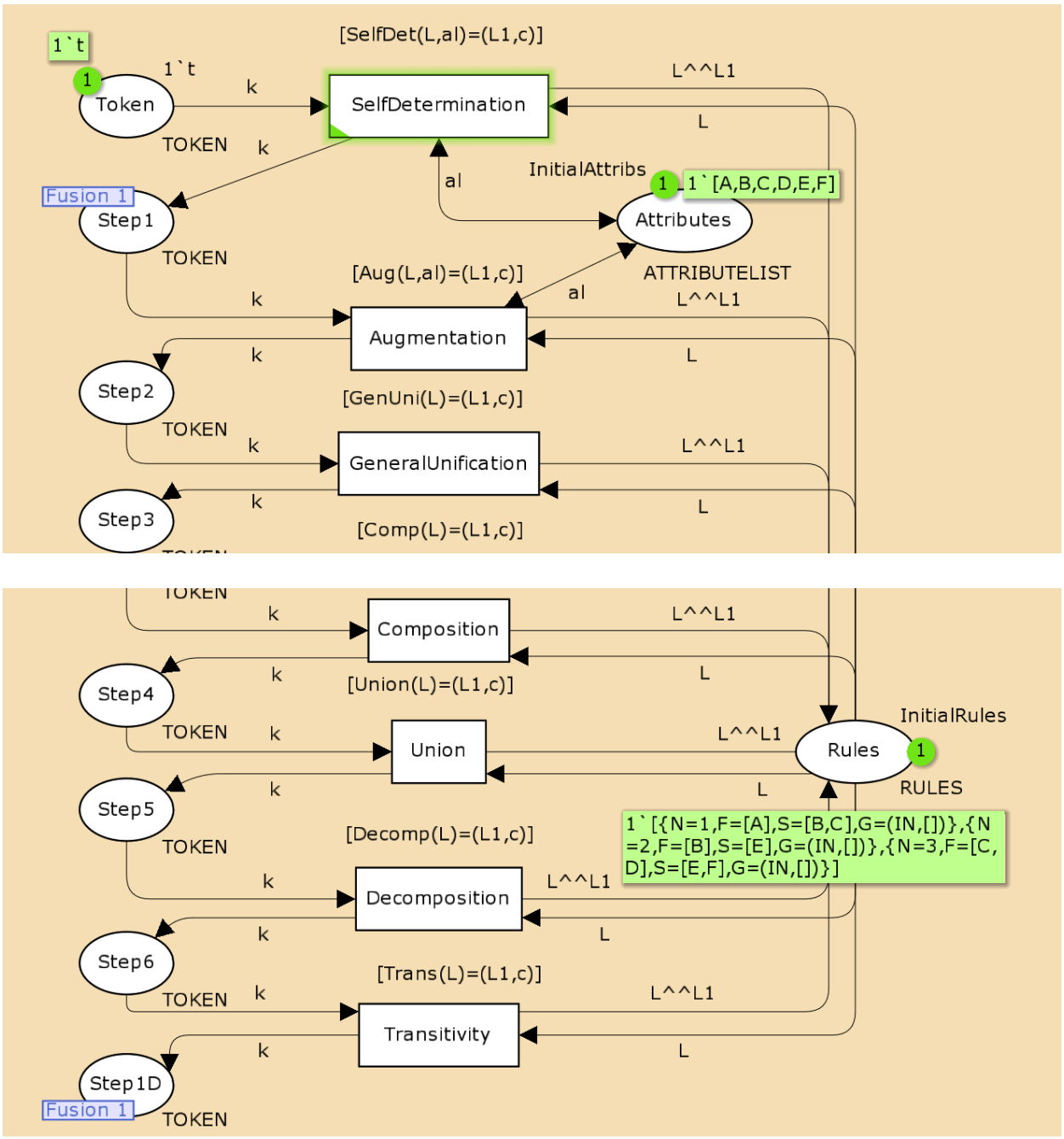
ایجاد مدل :

مدل مورد نظر مطابق شکل زیر ایجاد گردید . در مدل وضعیت های Step 1-6 **نشان دهنده ی مر حله به مرحله بودن اجرا می باشند . در هر کدام از** Transition **های موجود در مدل یک تابع از قوانین آرمسترانگ وجود دارند که با استفاده از مارکینگ های موجود در** Attributes **و** Rules **عمل می کنند .**

Arc **های یک طرفه نشانه ی خود را به هنگام اجرا شدن گذار خود مصرف و** Arc **های دو طرفه نشانه را مصرف نمی کنند .**

و وجود Token **به این دلیل است که زمان بندی بین مراحل وجود داشته باشد .**

Fusion **به این معنی است که هر دو وضعیت که** Fusion **هم نام دارند مارکینگ آنها نیز در هر لحظه یکسان می باشد .**



مدل ایجاد شده در Cpn Tools

توابع اولیه مدل :

```
fun getRuleIndex ( f: FD , (r::L): RULES ) : int =
if isEqual(#S f , #S r) andalso isEqual(#F f, #F r)
then 0
else
let val res = getRuleIndex(f,L)
```

```
in if (res <> ~1) then
res+1
else ~1
end
| getRuleIndex( _,[] ) = ~1;
```

این تابع یک FD **و یک لیست از** FD **ها را به عنوان ورودی گرفته و در صورت وجود** FD **در لیست عدد** 1 **و در غیر این صورت عدد** 1- **را بر می گرداند .**

```
fun isRuleExists( f: FD , L: RULES ) : bool =
let val n = getRuleIndex(f,L)
in if n <> ~1 then true
else false
end
| isRuleExists ( _ , [] ) = false;
```

این تابع همان کار تابع قبلی را منتبی با برگرداندن True **یا** False **انجام می دهد**

```
fun getAttribIndex( a : ATTRIBUTE, (r::L):ATTRIBUTELIST ) : int =
if a= r then0
else
let val res = getAttribIndex(a, L)
in if ( res <> ~1) then res +1
else ~1
end
| getAttribIndex( _,[] ) = ~1;
```

این تابع یک صفت و یک لیست از صفت ها را به عنوان ورودی می گیرد و در صورت عدم وجود صفت در لیست عدد -1 **و در غیر این صورت مکان صفت در لیست را بر می گرداند .**

```
fun isAttribExists( a : ATTRIBUTE, L: ATTRIBUTELIST ) :
bool =
let val n = getAttribIndex(a,L)
in if n <> ~1 then true
else false
end
| isAttribExists ( _ , [] ) = false;
```

این تابع یک صفت و یک لیست از صفت ها را به عنوان ورودی گرفته و در صورت وجود صفت true **و در غیر این صورت** False **را بر می گرداند.**

```
fun appendAttrib( a : ATTRIBUTE, L: ATTRIBUTELIST ) :
ATTRIBUTELIST=
let val exist = isAttribExists(a,L)
in if exist then L
else a::L
end
```

این تابع یک صفت و یک لیست از صفت ها را به عنوان ورودی گرفته و در صورت عدم وجود صفت در لیست آن را به لیست اضافه می کند .

```
fun difference(L1:ATTRIBUTELIST, L2:ATTRIBUTELIST) :
ATTRIBUTELIST=
```

```
let val n1 = List.length(L1)
val L = ref []
val i = ref 0
in

while !i< n1 do (
let val a = List.nth(L1,!i)
in if not( isAttribExists(a,L2)) then
if List.length(!L) = 0 then
L :=[a]
else L := !L ^^ [a]
else ()
end;
i := !i + 1 ); (* while i *)
!L
end
| difference( [], _) =[]
| difference( L , []) = L;
```

این تابع دو لیست از صفت ها را به عنوان ورودی می گیرد . و لیستی از تفاوت صفت های لیست 2 از لیست 1 را بر می گرداند .

```
fun merge(L1:ATTRIBUTELIST,L2:ATTRIBUTELIST):
ATTRIBUTELIST=
let val n2 = List.length(L2)
val L = ref []
val i = ref 0
in while !i< n2 do (
let val a = List.nth(L2,!i)
in if not( isAttribExists(a,L1)) then
if List.length(!L) = 0 then
L :=[a]
else L := !L ^^ [a]
else ()
end;
i := !i + 1 ); (* while i *)
!L ^^ L1
end
| merge( [], []) =[]
| merge( L , []) = L
| merge( [], L) = L;
```

این تابع دو لیست از صفت ها را به عنوان ورودی می گیرد و حاصل union **این دو لیست را بر می گرداند .**

```
fun isEqual(L1:ATTRIBUTELIST, L2:ATTRIBUTELIST): bool =
let val n1 = List.length(L1)
val n2 = List.length(L2)
val i = ref 0
val j = ref 0
val Found = ref true
in if n1 <> n2 then
false
else(
while !i< n1 andalso !Found do (
let val F1 = List.nth(L1,!i)
in j := 0;

Found := false;
while !j < n2 andalso !Found = false do(
let val F2 = List.nth(L2,!j)
in if F1 = F2 then Found := true
```

```
else ( )
end;
j := !j + 1 ) (* while j *)
end;
i := !i + 1) ; (* while i *)
!Found
)
end
| isEqual( [] ,[]) = true
| isEqual( _ , [] ) = false
| isEqual([],_) = false;
```

این تابع دو لیست از صفت ها را به عنوان ورودی می گیرد و در صورت یکسان بودن آنها مقدار `True` **و در غیر این صورت مقدار** `False` **را بر می گرداند.**

توابع بدیهیات آرمسترانگ :

```
fun Trans (L: RULES) : RULES * bool =
let val L2 = ref []
val n = List.length(L)
val i = ref 0
val j = ref 0
val cs = ref 0
val Found = ref false
val Gen = ref false
val nr = ref 0
in
nr := n+1;
while !i< n do (
let val F1 = List.nth(L,!i)
in j := 0;
while !j < n do (
if !i < !j then
let val F2 = List.nth(L,!j)
val ta = { N=(!nr), F = (#F F1),

S=( #S F2) ,G=(TR,[#N F1, #N F2])}
val tb = { N=(!nr), F = (#F F2),
S= (#S F1) ,G=(TR,[#N F1, #N F2])}
in if isEqual(#S F1 , #F F2) then
( cs :=1;
Gen := true )
else if isEqual(#S F2, #F F1)then
( cs := 2;
Gen := true )
else ( cs := 0;
Gen := false);
case (!cs) of
1 => if !Gen=true andalso
not(isRuleExists(ta,L))
andalso
not(isRuleExists(ta,!L2)) then
( L2 := !L2 ^^ [ta] ;
nr := !nr +1;
Found := true )
else ()
| 2 =>if !Gen=true andalso
```

```
not(isRuleExists(tb,L)) andalso
not(isRuleExists(tb,!L2)) then
( L2 := !L2 ^^ [tb];
nr := !nr +1;
Found := true )
else ()
|0 => ()
end
else ( );
j := !j + 1 ) (* while j *)
end;
i := !i + 1 ); (* while i *)
(!L2,!Found)
end
| Trans ( [] ) = ([],false);
```

این تابع دارای دو حلقه ی تو درتو که هر دو از ابتدا شروع می شوند می باشد . این تابع هر جفت از `Fd` **ها را بررسی می کند .اگر پیرو اولین قاعده برابر با صفت های ابتدایی قاعده دوم باشد . قاعده جدید می تواند با استفاده از قاعده ی انتقال آرمسترانگ نتیجه گیری شود .**

با تولید قاعده ی جدید شماره ی هر دو رول و قاعده ی جدید ثبت می گردد .

```
fun Aug(L: RULES, LA:ATTRIBUTELIST) : RULES * bool=
let val L2 = ref []
val n1 = List.length(L)
val n2 = List.length(LA)
val i = ref 0
val j = ref 0
val nr = ref 0
val Found = ref false
in
nr := n1+1;
while !i< n1 do (
let val F1 = List.nth(L,!i)
in j:= 0;
while !j < n2 do (
let val a = List.nth(LA,!j)
val t1 = { N= (!nr), F =
appendAttrib(a,(#F F1)),
S=appendAttrib(a,( #S F1)),

G=(AU,[#N F1])}
in if not(isRuleExists(t1,L)) andalso
not(isRuleExists(t1,!L2)) then
( L2 := !L2 ^^ [t1];
nr := !nr + 1;
Found := true)
else ()
end;
j := !j + 1) (* while j *)
end;
i := !i + 1); (* while i *)
(!L2,!Found)
end
| Aug ( [],_ ) = ([],false);
```

این تابع یک لیست از FD های جاری و لیست از صفت های یک رابطه را به عنوان ورودی می گیرد و قانون Augmentation نتیجه گیری می کند . این تابع قسمت ابتدایی و انتهایی یک وابستگی را با تمام قوانین موجود در پایگاه داده به ترتیب ترکیب می کند و در صورت وجود قانون جدید آن را به پایگاه اضافه می کند .

```
fun Decomp(L: RULES) : RULES * bool=
let val L2 = ref []
val n = List.length(L)
val i = ref 0
val Found = ref false
val nr = ref 0
in
nr := n+1;
while !i< n do (
let val F1 = List.nth(L,!i)
val Len = List.length(#S F1)
in if Len > 1 then(
let val t1 = {N= (!nr), F= (#F F1) ,
S= [ List.hd(#S F1)],G=(DE,[ #N F1])}
in if not(isRuleExists(t1,L)) andalso
not(isRuleExists(t1,!L2)) then
( L2 := !L2 ^^ [t1];
nr := !nr +1;
Found := true )
else ();
let val t2 = {N= (!nr), F= (#F F1) ,
S = List.tl( #S F1) ,
G=(DE,[#N F1])}
in if not(isRuleExists(t2,L))
andalso
not(isRuleExists(t2,!L2)) then
( L2 := !L2 ^^ [t2];
nr := !nr +1;
Found := true )
else ()
end
end
) else ()
end;
i := !i + 1 ); (* while i *)
(!L2,!Found)
end
| Decomp( []) = ([],false);
```

این تابع ق بررسی قاعده ی تجزیه آرمسترانگ می باشد .

این تابع تمام وابستگی های تابعی جاری را بررسی می کند اگر پیرو بک قاعده شامل یک صفت باشد از تابع خارج و در غیر این صورت قاعده ی جدید بر طبق قاعده ی تجزیه تولید می گردد . در صورت تولید قاعده ی جدید اگر آن قاعده در لیست از قبل وجود نداشت قاعده ی جدید به لیست اضافه می گردد .

```
fun Union(L: RULES) : RULES * bool =
let val L2 = ref []
val n = List.length(L)
val i = ref 0
val j = ref 0
val Found = ref false
val Gen = ref false
```

```
val nr = ref 0
in
nr := n+1;
while !i< n do (
let val F1 = List.nth(L,!i)
in j := 0;
while !j < n do (
if !i < !j then
let val F2 = List.nth(L,!j)
val ta = { N=(!nr) , F = (#F F1),
S=merge(#S F1 ,#S F2),
G=(UN,[#N F1, #N F2])}
in Gen := false;
if isEqual(#F F1 , #F F2) then
Gen := true
else ();
if !Gen=true andalso
not(isRuleExists(ta,L)) andalso
not(isRuleExists(ta,!L2)) then
( L2 := !L2 ^^ [ta] ;
nr := !nr + 1;
Found := true)
else ()
end
else ( );
j := !j + 1 ) (* while j *)
end;
i := !i + 1 ); (* while i *)
(!L2,!Found)
end
| Union ( [] ) = ([],false);
```

این تابع از دو حلقه ی تو در تو تشکیل شده است و تمام ترکیب های جفت `fd` **ها را بررسی می نماید اگر قسمت پیرو** `FD` **ها یکسان نبودند . آنها را با هم ترکیب می نماید .**

```
fun Comp (L: RULES) : RULES * bool =
let val L2 = ref []
val n = List.length(L)
val i = ref 0
val j = ref 0
val Found = ref false
val nr = ref 0
in
nr := n+1;
while !i< n do (
let val F1 = List.nth(L,!i)
```



```
in j := 0;
while !j < n do (
if !i < !j then
let val F2=List.nth(L,!j)
val t1={N=(!nr),F=merge(#F F1,#F F2),
S=merge(#S F1, #S F2),
G=(CO,[#N F1, #N F2])}
in if not(isRuleExists(t1,L)) andalso
not(isRuleExists(t1,!L2)) then
```

```
( L2 := !L2 ^^ [t1];
Found := true;
nr := !nr + 1 )
else ()
end
else ( );
j := !j + 1 ) (* while j *)
end;
i := !i + 1 ); (* while i *)
(!L2,!Found)
end
| Comp ( [] ) = ([],false);
```

این تابع از دو حلقه ی تو در تو که از ابتدا شروع می شوند تشکیل شده است . این تابع تمام جفت از `fd` **ها را بررسی و تمام صفت های قسمت ابتدایی را ترکیب می کند . و آن را به عنوان لیست ابتدایی قاعده ی جدید در نظر میگیرد . و همچنین قسمت پیرو هر جفت** `fd` **را هم به همین روش ترکیب میکند . و در صورت ایجاد قانون جدید آن را اضافه می نماید .**

```
fun GenUni(L: RULES) : RULES * bool =
let val L2 = ref []
val n = List.length(L)
val i = ref 0
val j = ref 0
val Found = ref false
val nr = ref 0
in
nr := n +1;
while !i< n do (
let val F1 = List.nth(L,!i)
in j := 0;
while !j < n do (
if !i < !j then
let val F2 = List.nth(L,!j)
val Dif21 = difference( #F F2, #S F1)
val Dif12 = difference( #F F1, #S F2)
val t1 = { N=(!nr), F = merge(#F F1,
Dif21), S= merge(#S F1, #S F2),
G=(GE,[#N F1, #N F2])}
in if List.length(Dif21) > 0 then
if not(isRuleExists(t1,L)) andalso
not(isRuleExists(t1,!L2)) then
( L2 := !L2 ^^ [t1];
nr := !nr +1;
Found := true )
else ()
else ();
if List.length(Dif12) > 0 then
let val t2 = {N=(!nr), F= merge(#F F2,
Dif12), S=merge(#S F1, #S F2),
G=(GE,[#N F2, #N F1])}
in if not(isRuleExists(t2,L))andalso
not(isRuleExists(t2,!L2)) then
( L2 := !L2 ^^ [t2];
nr := !nr +1;
Found := true )
else ()

end
else ()
end
```

```
else ( );
j := !j + 1 ) (* while j *)
end;
i := !i + 1 ); (* while i *)
(!L2,!Found)
end
| GenUni ( [] ) = ([],false);
```

این تابع نیز از دو حلقه ی تو در تو که در ابتدا شروع می شوند تشکیل شده است و با استفاده از فراخوانی دو تابع merge **و** difference **قانون** General Unification **آرمسترانگ را بررسی می کند**

```
fun SelfDet(L: RULES, LA:ATTRIBUTELIST) : RULES * bool=
let val L2 = ref []
val n2 = List.length(LA)
val j = ref 0
val Found = ref false
val nr = ref 0
in nr := List.length(L)+1;
while !j < n2 do (
let val a = List.nth(LA,!j)
val t1 = {N=(!nr), F= [a],S=[a],G=(SE,[])}
in if not(isRuleExists(t1,L)) then
( L2 := !L2 ^^ [t1];
nr := !nr + 1;
Found := true)
else ()
end;
j := !j + 1 );
(!L2,!Found)
end
| SelfDet ( [],_ ) = ([],false
```

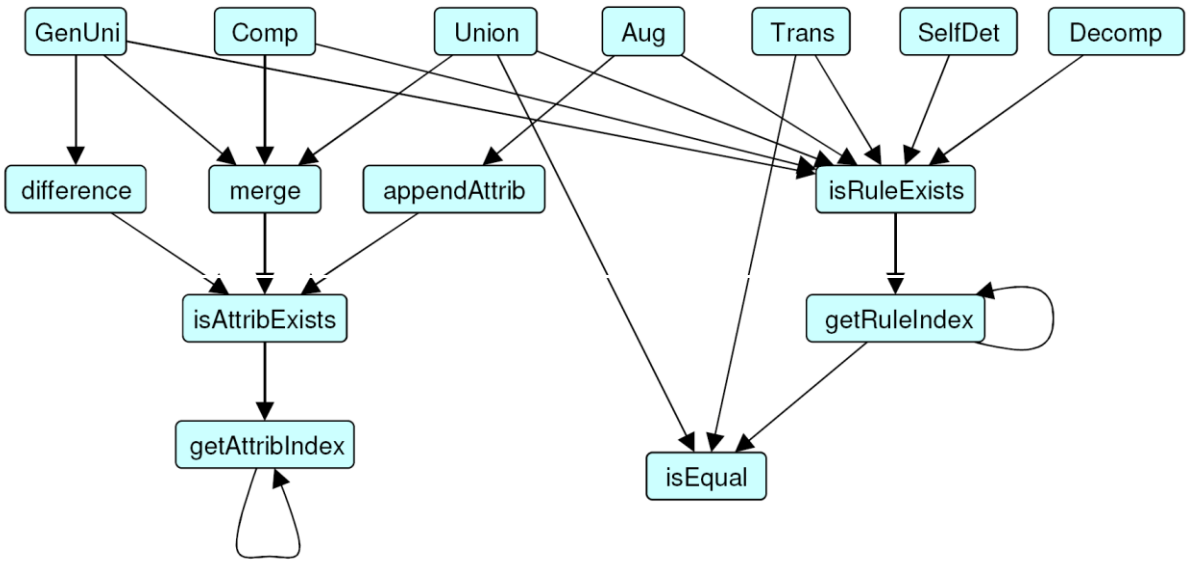
این تابع لیست قوانین و صفت های پایگاه داده را به عنوان ورودی گرفته و قانون self **آرمسترانگ را مورد بررسی قرار میدهد و در صورت عدم وجود قانون در پایگاه داده آن را اضافه می کند .**

تعریف توابع در نرم افزار باید به گونه ای باشد که توابعی که در توابع دیگر استفاده شده اند زودتر از آنها تعریف شده باشند .

```
fun getRuleIndex2
fun PRToStr
fun AtrToStr
fun ALToStr
fun isEqual
fun getRuleIndex
fun isRuleExists
fun getAttribIndex
fun isAttribExists
fun appendAttrib
fun difference
fun merge
fun Trans
fun Aug
fun Decomp
fun Union
fun Comp
fun GenUni
fun SelfDet
fun FDToStr
fun ExtractProof
```

تعریف توابع در Cpn Tools بدون جزییات

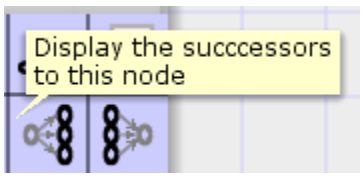
نمودار استفاده ی توابع اصلی از توابع ابتدایی :



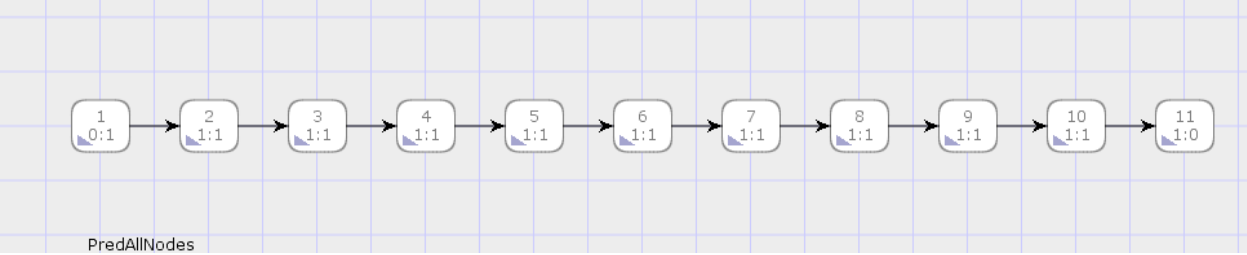
نمودار State Space

پس از ایجاد مدل با استفاده از ابزار State Space **در** Cpn tools **و با استفاده از گزینه ی** Save Report **به اطلاعات زیر در مورد مدل می رسیم که اطلاعاتی در مورد** State Space **شامل نود ها و آرک ها و وجود نشانه ها در وضعیت ها و ...ُمی باشد.**

```
 Statistics
------------------------------------------------------------------------
-------

  State Space
     Nodes:  15
     Arcs:   15
     Secs:   813
     Status: Full

  Scc Graph
     Nodes:  10
     Arcs:   9
     Secs:   0


 Boundedness Properties
------------------------------------------------------------------------
-------

  Best Integer Bounds
                              Upper       Lower
     Fd'Attributes 1          1           1
     Fd'Rules 1               1           1
     Fd'Step1 1               1           0
     Fd'Step1D 1              1           0
     Fd'Step2 1               1           0
     Fd'Step3 1               1           0
     Fd'Step4 1               1           0
     Fd'Step5 1               1           0
```



با استفاده از دو گزینه موجود در تصویر بالا می توانید نمودار State Space **حاصل از مدل را ایجاد نماییم.**



نمودار State Space

بررسی مدل و استخراج استدلال :

برای بررسی مدل نیاز به یکسری توابع داریم که در ادامه تعریف شده اند .(مشابه تعریف توابع اصلی درنرم افزار)

توابع بررسی مدل :

```
fun AtrToStr( atr : ATTRIBUTE ) : string =
case atr of
A => "A"
| B => "B"
| C => "C"
| D => "D"
| E => "E"
| F => "F"
| G => "G"
| H => "H"
| I => "I"
| J => "J";
```

تابع تبدیل صفت به رشته

```
fun PRToStr( pr : PRODRULE ) : string =
case pr of
IN => "Initial FD"
| SE => "Self-determination"
| AU => "Augmentation"
| GE => "General Unification"
| CO => "Composition"
| UN => "Union"
| DE => "Decomposition"
| TR => "Transitivity";
```

تابع تبدیل قوانین آرمسترانگ به رشته

```
fun getRuleIndex2 ( n: INT , (r::L): RULES ) : int =
if n = #N r then 0
else
let val res = getRuleIndex2(n,L)
in if (res <> ~1) then
res+1
else ~1
end
| getRuleIndex2( _,[] ) = ~1;
```

این تابع یک شماره و یک لیست از `fd` **ها را به عنوان ورودی می گیرد و در صورت وجود شماره در لیست موقعیت آن را بر می گرداند**

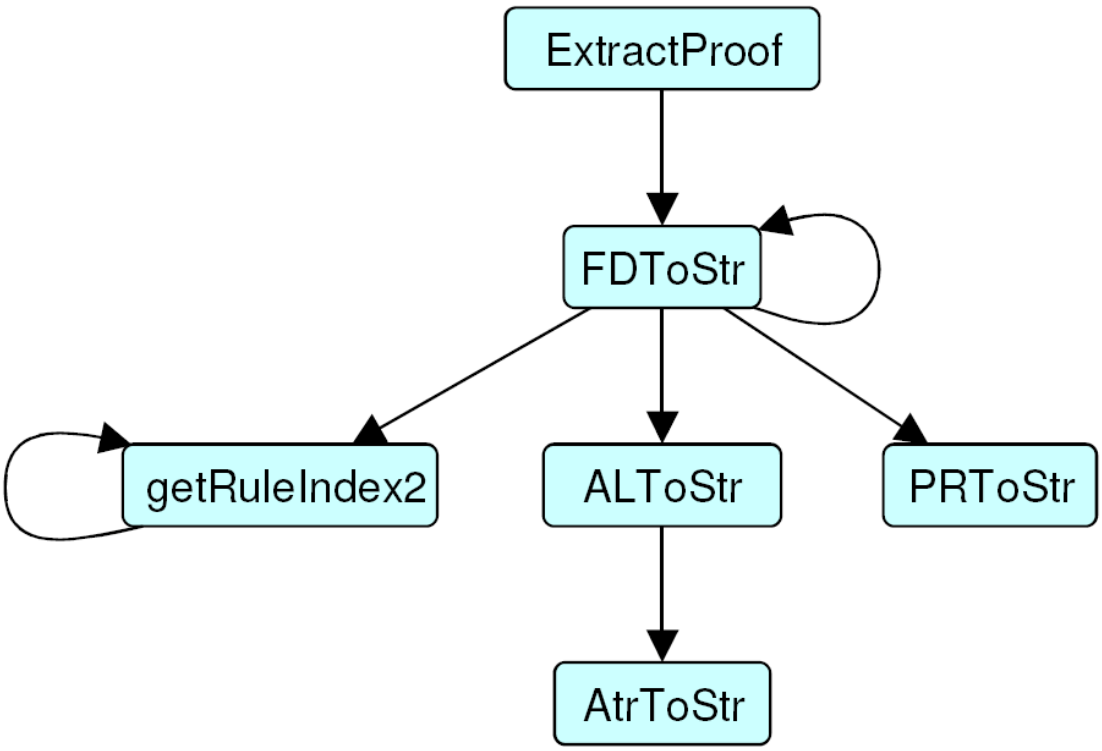
```
fun FDToStr( r: RULES, fd : FD ) : string =
let val st = ref ""
val s = ref ""
val i1 = ref 0
val i = ref 0
in
if #1 ( #G fd ) = IN then
st :=ALToStr( #F fd )^"-->"^ALToStr(#S fd)
else (
let val Len= List.length ( #2( #G fd) )
in while !i < Len do (
i1 := getRuleIndex2( List.nth( (#2
( #G fd)) , !i ) ,r);
s := !s^ FDToStr( r, List.nth(r, !i1));
i := !i +1;
if ( !i < Len ) then s := !s^","
else ()

);
st:="{"^!s^ "("^PRToStr( #1( #G fd )) ^") =>
"^ALToStr(#F fd )^"-->"^ALToStr(#S fd)^"}\n"
end );
!st
end;
```

این تابع یک `fd` **و لیست قوانین را می گیرد و آن** `Fd` **را به معادل** `String` **تبدیل می نماید .**

```
fun ExtractProof( r : RULES, n: INT ) : string =
let val ff = List.nth( r , n)
val s = ref ""
val f = TextIO.openOut "Proof.txt"
in s := FDToStr( r, ff );
TextIO.output(f, !s);
TextIO.closeOut f;
!s
end
| ExtractProof ( [] , _) = "";
```

این تابع قوانین و شماره ی `FD` **را به عنوان ورودی می گیرد و حاصل استدلال آن** `FD` **را نمایش و در فایل** `proof.txt` **ذخیره می نماید**

نمودار استفاده ی توابع از



توابع استخراج استدلال :

با استفاده از توابع زیر State Space **را مورد بررسی قرار می دهیم .**

ما دنبال این هستیم که ببینیم آیا مدل ایجاد شده ی ما با FD **های ابتدایی آیا قادر به نتیجه گیری قانون** AD->F **می باشد ؟**

```
fun findNodes n = ( isRuleExists( finalFD , ms_to_col
(Mark.FDClosure'Rules 1 n) ) = true);
```

این تابع یک نود از State Space **و** FD **هدف را می گیرد و اگر آن** FD **در آن نود ظاهر شده بود** True **و در غیر این صورت مقدار** False **را بر میگرداند**

```
val findNode = fn: Node -> bool
```

```
PredAllNodes findNodes;
```

تمام نود هایی State Space **را که شامل** FD **می شوند را در یک لیست بر می گرداند .**

```
List.hd (PredAllNodes findNodes)
```

نود ابتدای لیست حاصل از دستور آرگومان را بر می گرداند

```
ms_to_col (Mark.FDClosure'Rules 1 (List.hd (PredAllNodes findNodes))
```

لیست قوانین موجود در نود آرگومان را بر می گرداند یعنی نود 9

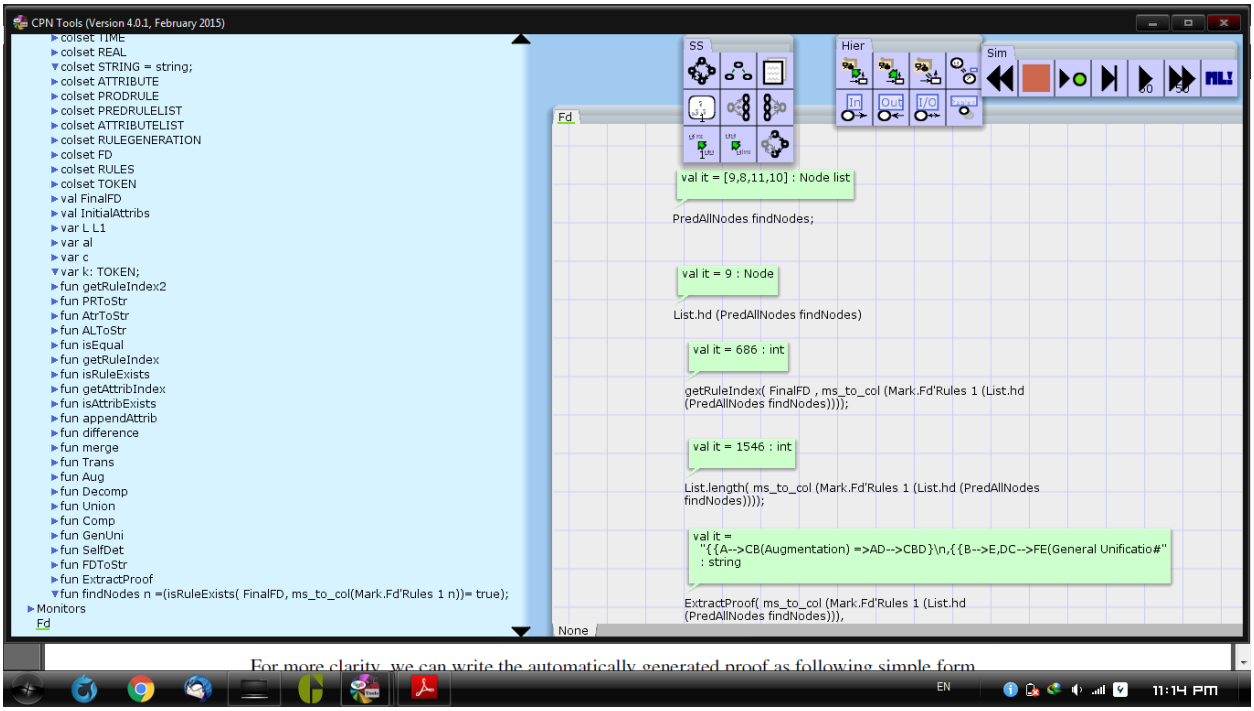
```
getRuleIndex( finalFD , ms_to_col (Mark.FDClosure'Rules 1 (List.hd
(PredAllNodes findNodes))));
```

این ml code **شماره** FD **هدف ما را در نود شماره ی** 9 **بر می گرداند**

```
ExtractProof( ms_to_col (Mark.FDClosure'Rules 1 (List.hd
(PredAllNodes findNodes))),
getRuleIndex( finalFD , ms_to_col (Mark.FDClosure'Rules 1 (List.hd
(PredAllNodes findNodes)))));
```

و این تابع هم همانگونه که در شکل دیده می شود نتیجه ی استدلال که حاصل آن FD **مورد نظر گشته را بر می گرداند .**

برای هر تابع در فضایی از Net **مطابق شکل زیر یک و با ایجاد** Aux Text **و نوشتن تابع در آن و سپس با فشردن کلید** Evaluate Ml **از ابزار** Simulation **در نرم افزار به نتیجه ی زیر رسیدیم . که نشان دهنده ی استخراج خودکار برای** Final Fd **می باشد .**



بعضی از توابع این قسمت که از State Space **استفاده می کنند باید پس از فشردن کلید محاسبه ی نمودار** State Space **از ابزار** State Space **نوشته شوند .**

نتیجه گیری :

تولید استدلال خودکار به مدیران پایگاه داده برای نرمال سازی پایگاه های داده کمک می کند .

شبکه های پتری رنگی یک ابزار قدرتمند برای بررسی و تحلیل برنامه های کاربردی است .

مدل ایجاد شده نشان داد که می توان با استفاده از CPN **استدلال خودکار برای تولید قوانین وابستگی های تابعی را داشت.**

کاهش هزینه در استدلال خودکار می تواند مورد مطالعه و پژوهش بیشتر قرار گیرد .