NORMA August 2024 Release

Unary Negation, Absorbed Subtype Columns, Default Values

Unary Negation

The NORMA August 2024 release includes a significant change to unary fact types. This is a one-directional change, so if you open and save any previous .orm file (with at least one unary fact type) then you will not be able to load that file in previous NORMA versions. Although I recommend using automatic updates for the NORMA extension, if you have a co-modeler who does not do this then you should encourage them to update to avoid friction in your working relationship.

Old Unary Fact Types (Binarized)

NORMA has long used a fully binarized form of the model hiding just under the surface. With this approach, the model has a live parallel representation using functional binary fact types (one single role uniqueness constraint) or one-to-one fact types (two single role uniqueness constraints). Ignoring the non-binarized form (n-ary fact types, objectifications and spanning-binary fact types) is easily done, as is using the parallel binary form for analysis. The binarized form is a powerful tool to simplify mapping to physical models by tightly limiting the number of fact type patterns that mapping algorithms need to consider.

As part of this binarization push, we also (circa November 2006, when NORMA was in its infancy) binarized unary fact types by implicitly creating an implied Boolean value type for each unary fact type. This felt like a good idea at a time. However, the issues with this early decision became increasingly clear over time.

1. Almost all uses of unary fact types needed special consideration in the NORMA code. For example, Fact Type Shapes were always given a role order (with one role) to force the underlying value role from displaying, Uniqueness and Frequency constraints were attached to the hidden role while all other constraints attached to the visible unary role, and link fact type patterns used a unique role subtype (ironically, so that reducing to a binary-only form kept the objectified unary role instead of removing it like all other objectifications).
2. In addition to the special cases for unaries, the project was also sprinkled with code to determine if a binary was really a binary fact type or a unary masquerading as a binary.
3. It turned out that special casing was required practically anywhere these binarized unaries were consumed, effectively negating the advantage of the binarized form.
4. Logically, although true and false were allowed, there was no natural way to reference the false state, so constraints and objectifications were assumed to apply to the ‘true’ state. For example, if you placed a disjunctive mandatory constraint on a unary role (and

another role to make a disjunctive mandatory constraint) the unary fact type only satisfied the constraint when the population of the implied role was true, not false.

5. This meant that unary fact types did not follow the same role population semantics as all other fact types. For example, if a $persion_1$ plays the Person role in *Person has PersonName* then you know $person_1$ has a name. However, if $person_1$ plays the Person role in *Person is married*, then you cannot assume $person_1$ is married without checking the *true* state of the implied role.
6. This emphasis on the *true* state (when both *true* and *false* were possible values) seriously neglected the *false* state. There was no way to incorporate the false state into a constraint—such as a subset constraint when the unary is known to be false. There was also no way to objectify the false state: objectification had to assume true because using one object to objectify polar opposite states is not meaningful.
7. We could not make a unary fact type mandatory—to force a true/false choice—because a unary fact type cannot be mandatory. This was an odd state where we were stuck between the unary display (where mandatory makes the fact type worthless because it implies all instances play the role, so there is no reason to record it) and the underlying binary representation. A simple mandatory constraint here would have had a different meaning, so could not be displayed.
8. The negation of a unary in logic was not exact because of the 3-state system (all unaries were mapped to nullable true/false). So, *for some x of type X where x.a* is meaningful and exact—the unary has the true value recorded. However, *for some x of type X where ~x.a* is ambiguous because we cannot distinguish between the false and null states, ~x.a is true for both states. There is simply no way to convey that 'x.a is known to be false'.

So, effectively, we made a mistake in the dawn of NORMA. The live binarization has proven to be extremely useful for all constructs—except unary fact types. Unary fact types have been less powerful than they should be, are always mapped to true/false/null in a relational mapping (even when false is not used) and must frequently be special-cased even when analyzing the fully binarized form of the model—thus negating the benefits of binarization in the first place. A unary is simply not difficult to map compared to other complex forms (an objectified quaternary fact type with a surrogate id, for example), so there is minimal benefit in standardizing with a construct that does not map well to the underlying logic and needs to be special cased most of the time regardless.

Unary Negation Fact Types

From this point forward, a unary fact type is just a unary fact type. There is one role. There is no binarized form. There is no implied value type. If an instance plays a unary fact type, then there is no need to check if the opposite value is true. Simply playing a unary role now has the same semantics as playing any other role.

Clearly, this is the easy approach for logic, and ORM is a graphical representation of first order predicate logic. However, ORM is also a domain modeling tool, and it is extremely valuable in the domain modeling space to have a single element that can represent a true/false (or yes/no) state. Basically, any fact type that maps to the answer to a yes/no question is likely to have multiple states. For example, *Do you prefer the new unary fact types?* has a natural answer set of yes/no/unknown—or true/false/null (hopefully, you'll give an unequivocal 'yes' answer by the end of this discussion).

To model this natural domain fact type without a *false* state would require two unary fact types and an exclusion constraint across the roles (or exclusive or constraint for the a required answer) constraint. However, even this clumsy construct is not sufficient to classify these as formally opposite states because in a physical mapping it will be two columns (properties, attributes, etc.) The value for each column will be 1, and it is not clear (without adding a very smart language parser for the readings) which of these exclusive unaries is the 'true' state and which is the 'false' state. Clearly, two unaries with an exclusion constraint is not sufficient on its own to accurately describe a true/false/unknown answer in a model representing a natural-language survey question.

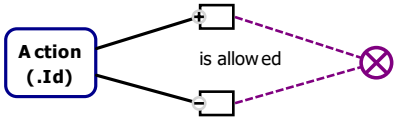
To resolve this tension between the language and logic views of the model, we have added a *unary negation fact type* to NORMA. This fact type is formally related to a positive (aka normal) unary as its negation. This fact type does not have to be seen by the end-user at all—unless there is a need to constrain or objectify the *false* state. In addition to the negation fact type, the construct also creates an exclusion constraint and possibly a mandatory constraint. To simplify the possible combinations of unary fact types and associated constraints—and to ensure they are all correct when defined—this pattern is managed through a new *UnaryPattern* property offered on a unary fact type.

The *UnaryPattern* property has three main states based on combinations of possible True, False and Unspecified values. The term *Unspecified* term was chosen over *Null* (too closely associated with physical implementations) and *Unknown* (I started with Unknown, but I consider a null statement to be a known state, so I did not like the implication that a null state is the same as an unknown state). Using *Unspecified* is a few more letters, but I believe this is the best representation of the conceptual intent, allowing for the user to associate the lack of a fact type with a meaning.

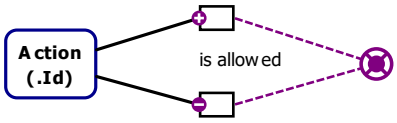
The four main *UnaryPattern* states are:

1. *True/Unspecified* indicates that there is no negation. This is a simple, non-negatable unary fact type. These are now physically mapped to only allow true values (or null).
2. *True/False/Unspecified* indicates a paired negation unary with a pair exclusion constraint. The exclusion constraint is included in the model (and placed inside the positive unary in the Model Browser) and can be placed on the diagram if desired by dragging it from the model browser. The full structure is shown here. A reading for

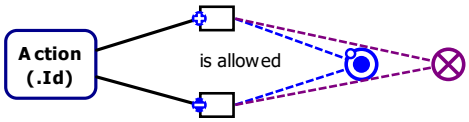
the negation is not required (unless either unary is objectified) but would generally be added for clarity if the shape is shown. However, the negation only needs to be displayed if it is objectified or involved in constraints. Otherwise, the plus on the primary unary indicates that a negation is available (and the minus indicates that this is a negation).



3. *True/False* indicates that one of the pair must be specified. The display is changed to:



4. A final option makes the mandatory deontic and is shown as *True/False obligatory*. This displays as:



   Note that the dot is notched at the four ±[3]π/2 locations. This gives a slight distinction for printed forms of the model. It is expected that this form will be the least-used, especially with the availability of default values (discussed next).

In practice, the constraints will not be displayed, and the negations will not be shown unless needed for objectification or constraints. The state of the + annotation on the positive unary, which can have one of the three shown forms for a negatable unary, is sufficient to indicate the entirety of each pattern. Generally, if both fact types are shown, proximity on the diagram and a provided reading for the negation are sufficient to indicate the unary pairing. The unary fact type verbalization also includes the implied exclusion (and possible mandatory). For example, the *True/False* unary pattern verbalizes the positive unary fact type as follows:

```
Action is allowed.
    In each population of Action is allowed, each Action occurs at most once.
    For each Action, exactly one of the following holds:
      that Action is allowed;
      that Action ~is allowed.
```

The verbalization also shows the implied reading for a negation unary (*A r* becomes *A ~r*, and *r A* becomes *~(r A)*). Obviously, if a reading is supplied for the unary negation fact type then the default negation reading is not allowed.

The *UnaryPattern* property also allows the modeler to specify default values. So, for example, you'll see *True/False (default True)* listed just below the *True/False* value in the list. The default values are filled into the DefaultValue property as *True* for the matching unary role—a *default*

False state becomes a True DefaultValue on the role for the negation unary. For a unary fact type these defaults can only be controlled through the UnaryPattern property, so are presented as read-only. As with other default values, if you have a unary on an absorbed object type (subtype, unary objectification, etc.) then the database cannot include the default.

Unary Negation Editing Gestures

The *UnaryPattern* property on a unary fact type is the controlling mechanism for the supported unary negation patterns. This property is available on both the positive and negation unary sides of the fact type pair. *UnaryPattern* is also an emphasized property on a unary objectification, so you can see and set it without using the expandable *ObjectifiedFactType* property. The *UnaryNegation=True* read-only property is also set if the negation fact type is selected.

The *UnaryPattern* is the only mechanism available for setting default values, *True/False obligatory* state, and for removing the unary negation. However, other editing gestures are integrated in other parts of the tool to make them easier to work with.

Is Mandatory Menu

The *Is Mandatory* menu is enabled on any unary fact type. If the unary has no negation or allows an *Unspecified* state, then selecting this will jump directly to a *True/False* unary pattern. If *True/False* or *True/False obligatory* are set then the menu item is checked and selecting it will drop back to a *True/False/Unspecified* pattern. Default value settings are preserved.

This *Is Mandatory* menu item is not the same as the *IsMandatory* property in the Properties Window for a selected unary role. The role property will always be False and read-only for a unary fact type. The *Is Mandatory* menu is hijacked for convenience and refers to the disjunctive mandatory on the paired unaries.

Fact Editor ~ Notation

The tilde (~) character is frequently used in logic expression to represent the *logical not* operator. Therefore, since the positive fact type and its negation are logical inverses of each other, this is a good notational match to represent 'the other paired unary fact type' when one is selected. Therefore, it is appropriate to use in the implied negation reading of a positive unary (A ~r) and in the fact editor, which displays readings based on selection.

Note that ~ is not very useful in shape display, as discussed earlier, because the shape needs context from some other annotation. Since ~ means the *NOT* operator, it cannot on its own imply if the thing it is operating on is positive or negative, so ~Ar will mean the negation if r is the positive reading—or the positive unary if r is the negation reading (and the unary pattern is True/False so the null state is not in play).

The ~ character is used with a unary fact type in the Fact Editor similar to the way the forward slash (/) character is used to specify forward and reverse readings with a binary fact type. The behavior is subtly distinct, however.

1. The ~ specifies the start of the inverse reading, so *Action is allowed~is disallowed* specifies both the positive and negative forms for a new fact type. However, if the negation unary is selected, then this will show as *Action is disallowed~is allowed.*
2. The inverse reading is not required to use the ~, so the *Action is allowed~* will create a negatable unary with no negation reading. This can also be done after the non-negatable unary is created (select the unary, add ~ in the Fact Editor, Ctrl-Enter to commit the change).
3. Since unary fact types have one role, the ~ can work on either side of the named role player, so *disallow~allow Action* is valid.
4. Using a tilde in the reading will force a negatable state (True/Unspecified will become True/False/Unspecified) but the fact editor cannot be used to remove the negation. Removing a negation requires a change to the *UnaryPattern* property. So, changing *Action is allowed~is disallowed* to *Action is allowed~* or *Action is allowed* will not affect on the unary pattern (the first with the trailing tilde will remove the *is disallowed* reading, and the second will not change the inverse reading).

Reading Editor Additions

Selecting a negatable unary in the reading editor will add an extra branch. This structure is similar to the Reading Editor behavior when selecting a role in an objectified (or spanning binary) fact type. In the objectification case, selecting a role will show *Fact Type Readings*, which are the readings for the selected fact type, and *Implied Fact Type Readings*, which are the readings for the link fact type corresponding to that role.

For a negatable fact type selection, you will see *Unary Negation Readings*, which contains the readings for the paired unary. This header will change to *Positive Unary Readings* if a role in a negation fact type is selected.

If the unary is objectified and negatable both the implied fact type and negation/positive headers will appear at the same time.

Shape Drag Operations

If either side of a paired unary (or the unary role) is selected, then you will see a *Drag Unary Inverse* menu item. Clicking this will create a drag cursor for the paired fact type, which you can then click onto the diagram. This is effectively a shortcut for selecting the fact type in the model browser, locating the paired fact type, and dragging it out. (You will still need to do this to see the constraint shapes).

I liked this construct, so I added similar drag operations in two different places:

- The context menu for a role in an objectified (or spanning binary) fact type has a *Drag Link Fact Type* menu item. As with *Drag Unary Inverse* (which may also be there), this is a shortcut for tracking down the link fact type in the model browser and dragging its shape out.
- The Fact Editor now has two commit commands. The traditional one (auto populate new shapes) is still bound to Ctrl-Enter, but a new Alt-Enter command is also available to drag the shapes. You can see both of these on the new toolbar to the left of the Fact Editor (along with shortcuts if you select the Fact Editor window and hover over the toolbar).

  The new command lets you click exactly where you want your shape to go. This will create the committed fact type shape only—you can drag the roles out to create the object type shapes. This is very handy if you are adding to a complex diagram, where the auto placement can be haphazard (at best).

  The Fact Editor also supports adding a listing of objects if no reading text is specified. In this case you will get one click for each object. For example, if you click the diagram (for an empty window) and type SubAction VirtualAction, then Alt-Enter, you will now have two clicks available—a shape is placed first for SubAction, then the second click will place VirtualAction.

  If you escape out of drag mode the committed elements will still be in the model, just not on any diagram. This differs slightly from *Ctrl-Enter*, where the model change and diagram change occur in the same undo item.

World Assumption

I chose to use *UnaryPattern* (a term of my own making) instead of the notion of *world assumption* discussed in Dr. Halpin's seminal ORM book. In his presentation (section 10.6), he discusses 3 world assumptions (presented as CW, OW and OWN, although I believe the latest edition changes CW to CWN to indicate the false value is stored):

- **C**losed **W**orld: If not stated as true, the fact is false.
- **O**pen **W**orld: If not stated as true, the fact is unknown.
- **O**pen **W**orld (with **N**egation): The fact can be declared to be both *known to be true* or *known to be false*

In Dr. Halpin's presentation, *WorldAssumption* aligns with *UnaryPattern* as follows:

- CW = True/False
- OW = True/Unspecified
- OWN = True/False/Unspecified

Clearly, world assumption does not cover all of the unary patterns (True/False obligatory and the default value states), so it not simply a drop-in replacement without introducing additional meta data to the ORM model. Implementing world assumption would also have had to address the same issues as we tackled with unary negation (paired unaries to address objectification and constraints), so would have led to the same underlying structure.

However, my primary objection to the *world assumption* is the assumption part. In Dr. Halpin's discussion, the assumption is applied in the relational mapping, so CW=a non-nullable bit column (0 or 1). The problem with this approach is that world assumption describes the *interpretation of the data*, but the interpretation of data cannot be enforced by the how the data is stored. One of the first lessons in ORM is that the data itself is not sufficient to describe meaning. So, True/False (1 or 0) in the database and True/Unspecified (1 or null in the database) can clearly both be interpreted as true/false values.

It is frankly arbitrary how the unary state is stored as long as the application (business logic, UI etc.) knows what the data means. In other words, I can separate storage and meaning. If I'm in a rebellious mood—or more likely just prefer a nullable column so my database is easier to populate—I can interpret 1/null as CW and 1/0 as open world. It frankly doesn't matter how it is stored *as long as the application knows what it means*.

By introducing *UnaryPattern* I avoided the whole non-conceptual question of where an assumption should be applied (storage, business logic, UI, etc.) and simply present the possible underlying states of the data. With *world assumption* patterns I also found myself mentally translating them to the possible states to process what they meant. The presented unary patterns show the possible states directly, allowing me to skip a mental step to read the model.

New Relational Columns

Another issue with 'classic' unary fact types is that they have mapped similarly to subtypes but were never close enough to simply interchange a unary fact type and a subtype that plays no roles. These are conceptually very close. Even though a subtype with no played roles is generally considered poor conceptual form, it is definitely valid ORM. However, the problem with interchanging these constructs was a result of the relational mapping more than the conceptual issues.

- The unary would map to a nullable bit column (0 or 1), even though 1 (without the zero) would generally be the only used value.
- A subtype with no role players could not be absorbed because this left no evidence of the subtype in the database schema, so the *AbsorptionChoice* property was ignored and these objects always pushed to a separate table.

This really was not an acceptable implementation. It would clearly be better if these two constructs mapped to a consistent structure so that conceptual preferences would not affect the final mapping.

The next question with a subtype—and this applies to an objectified unary fact type as well—is whether the bit value is even needed to indicate the absorbed subtype. The question of *is a supertype an instance of one of its (absorbed) subtypes* falls into 3 categories:

1. The 'is of subtype' answer is fully derivable, which occurs if the subtype plays mandatory roles. The mandatory constraints can be simple, or with all roles in a disjunctive mandatory played by the subtype.
2. The 'is of subtype' answer is partially derivable. This occurs when the subtype does not have mandatory roles but still plays some roles. In this case, conceptually, the subtype can either be inferred—if a non-mandatory role is played—or declared explicitly.
3. The 'is of subtype' answer is fully asserted. In this case, no roles are played by the subtype, so it can never be inferred.

From a mapping perspective, this means that for case #1 there is no need to store an 'is of SUBTYPE' column, but we need a way to assert the subtype for cases #2 and #3. For an objectified unary, this will also mean that the bit column is not needed at all if the unary objectification object type plays a mandatory role.

This question of whether or not an extra column is needed to satisfy a declared subtype (or if the unary column is *not* needed) is based on analysis of the mandatory state for all absorbed types. If the type is separate (which is available for both objectified unary types and subtypes) then this analysis is not needed. However, for an absorbed type, any absorbed subtype that cannot be fully derived needs data to declare the supertype as an instance of the subtype. We refer to these as *isSUBTYPE* columns, but the same pattern applies to other absorbed 1-1 constructs (the link fact type for a unary objectification is 1-1). Note that if the mandatory constraint is on a non-functional role (one without a single-role uniqueness) then subtype evidence can be in another table, but the most common case of mandatory constraints on functional roles will mean the data to infer the subtype is available in the same row.

Unary vs Objectified Unary vs Subtype

The relational mapping for a unary negation is pretty clear—a single column is created with a Boolean (0/1 bit in SQL) value. This paired column will be non-nullable if the unary pattern is *True/False* and if the unary is not part of an optional absorbed object type (usually a subtype or objectification). With this type of column, the 0 state represents that the negation fact type is populated. A true-only column (0 is not allowed) is created for a non-negatable unary. None of this is difficult.

The trickier part of the mapping comes when either (or both) of the unary fact types is objectified. In this case, the unary pairing is broken. For example, if a positive unary is objectified and the negation is not objectified, then the negation is mapped as a standalone true-only column and the objectified value is not mapped to a column (either the 0 or 1 state). By default, the objectification will be absorbed, so the mandatory-analysis will be applied to determine if an extra column is needed to assert the subtype.

If the extra column is needed, then we will check if this comes from a unary pair and if the opposite unary also needs a column (so is not objectified, or objectified and requires a maker column). We use the *unobjectified* naming analysis for these columns, so the column name will look the same if it is paired directly *or* if it is being treated the same as an *isSUBTYPE* column. However, the analysis path to get this extra column paired and named is completely different than the case with no objectification.

The old NORMA unary structure used a specialized subtype of Role for unary objectification. In the standard objectification, all mapping is done over the link fact types and the original roles are eliminated. In the unary case, however, the objectification was aligned with the *true* state, but the *false* state was always available as well. This meant that we generated a column for the unary fact type in both objectified and unobjectified cases to make sure the false could be stored, thus representing both the link fact type and the objectified role in the mapping—unlike all other objectifications where only the link fact types are mapped. This extra complexity was removed with unary negation because the two states are separate fact types that are opportunistically combined into a single column. Unary objectification now has the same meta-model as any other objectification.

isSUBTYPE Columns

As a result of the new mandatory-constraint analysis, an absorbed subtype (or absorbed objectified unary fact type) will now get an extra column called *isSUBTYPE* in the primary table. Previously, we ignored the *absorb* request if there was no evidence—meaning extra columns—in the absorbing table. However, this meant that there was no distinction between the fully-derived and partially-derived states.

The new mapping is far superior and accurately reflects the model. If it is determined that a column is needed (for asserted or partially derived subtypes) then the new column will be called *isSUBTYPE*, where *SUBTYPE* is the type name (with relational naming rules applied). For unary objectification, the *isSUBTYPE* column uses the unary fact type naming instead of *isOBJECTIFIED_TYPE_NAME*, but under the covers these are the same as *isSUBTYPE* columns used for a subtype.

IsIndependent Availability

Although an objectified unary fact type and a subtype have a similar mapping, the appearance of *isSUBTYPE* columns may be hard to predict because of two rules in ORM theory:

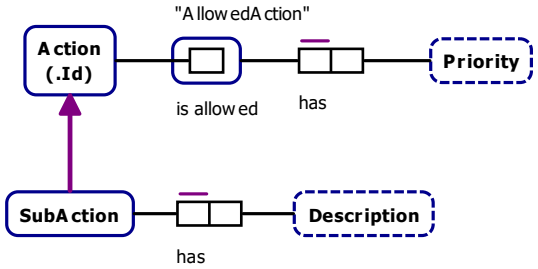
1. An object type that plays no non-identifying mandatory roles has an implied mandatory constraint across all played roles *unless* the object type is marked as independent.
2. The supertype relationship is considered mandatory, so a subtype can never be independent.

In practice, this means that if a subtype plays a single non-mandatory role, then there is no implied mandatory constraint on that role. However, if a unary objectification plays a single non-mandatory role, then that role is implicitly mandatory *unless* the objectifying type is marked with *IsIndependent=true*. This might be a little confusing at first: the subtype will get the *isSUBTYPE* column, but the unary will not show the subtype marker column unless it is independent.

This demonstrates the tight mapping between independent object types and *isSUBTYPE*-equivalent columns.

- If an absorbed type can be independent, then it will only have an *isSUBTYPE* column if it either plays no roles or is marked with *IsIndependent=true.* Generally, an enabled *IsIndependent* property means the object type not a subtype, so an absorbed type that can be independent is usually the objectification of a unary fact type, although direct 1-1 identification of any object type can also cause an absorption attempt into the identifying table
- A subtype will never have an implied mandatory constraint, so an absorbed subtype will have an *isSUBTYPE* column unless the subtype explicitly plays a mandatory role.

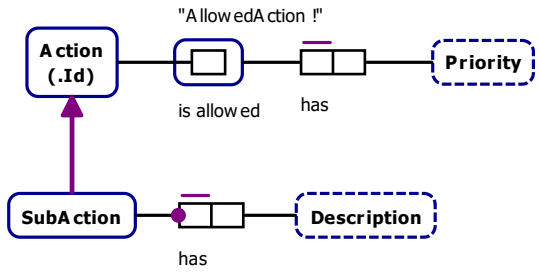
For example, with this model:



You'll see the following relational table with an *isSubAction* column because SubAction does not play a mandatory role, but no *isAllowed* column because *AllowedAction has Priority* is implicitly mandatory.

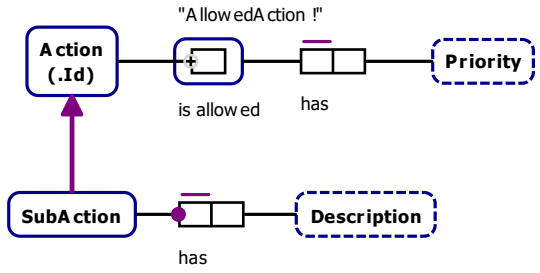
Action
PK : actionId : int
allowedActionPriority : int
isSubAction : true
subActionDescription : varchar(512)

When we modify the model by making *AllowedAction* independent and adding an explicit mandatory constraint we gain a column for AllowedAction and lose one for SubAction:



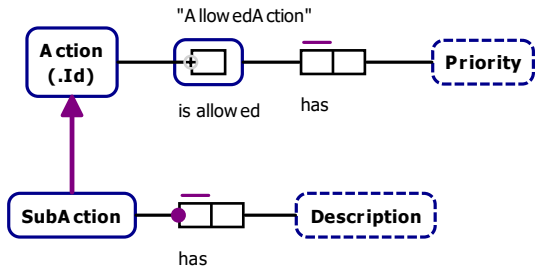
Action
PK : actionId : int
allowedActionPriority : int
isAllowed : true
subActionDescription : varchar(512)

Adding an negation to the unary (select *is allowed*, change the fact editor to *Action is allowed~is disallowed* and commit) pairs the negated state with the isSUBTYPE-style column for isAllowed, giving a Boolean type.



Action
PK : actionId : int
allowedActionPriority : int
isAllowed : boolean
subActionDescription : varchar(512)

Finally, if we remove the independent state, the extra type column is no longer needed, so the unary negation can no longer be paired. Since *is disallowed* can longer be represented as the 0 value in the *isAllowed* column, it gets its own *true*-only column. This is the same mapping as two separate unary fact types (instead of a unary negation). This also demonstrates why unary negations require a reading if either state (positive or negation) is objectified.



Action
PK : actionId : int
allowedActionPriority : int
isDisallowed : true
subActionDescription : varchar(512)

This difference in implicit mandatory semantics also means that if you *want* the implied mandatory behavior on a subtype, you can use an objectified unary instead of the subtype and obtain the same mapping, at least in the relational mapping—other physical mappings may require a representation of the objectified type. Regardless of your conceptual preference, both subtypes and objectified unaries can now be an easier representation than multiple individual subset constraints: create a subtype (or objectified unary) with a mandatory role player, then attach your subset role players to the same type to conceptual group the fact types with no side-effects in the mapping. Also, NORMA does not currently relationally map the external subtypes, so in addition to removing a lot of external-constraint noise from the model, this will give you stronger check constraints in the generated DDL.

Migration from Earlier NORMA

The first time you open NORMA after upgrading you'll see a message describing these changes with links to this white paper and an overview video on the same topic. This is meant as a warning that any older model you open with your new NORMA will require review of the unary fact types.

The default unary conversion from the binarized form to the unary negation form preserves the old mapping, which is a unary pattern of *True/False/Unspecified*. This means you'll see all of the unary fact type shapes displayed with the circled-plus notation.

The easiest way to review these is to use the *Fact Types* expansion in the *ORM Model Browser* tool window. The unary fact types can be easily spotted here. I dock the Properties window below the model browser on the right of the work area and the *ORM Fact Editor* tool window below the diagram. You can now select a unary fact type, determine the appropriate unary pattern, and easily add a reading for the negation if you choose to keep the negation. You can

also review the verbalization and select the fact type in a diagram if you want to make further changes.

After review—with or without unary changes, but definitely with them—you may have a modified DDL file. Modifications will including stricter types on the unary columns (true-only restrictions) and new *isSUBTYPE* columns. You will need to upgrade your data to match these changes. For unaries, there are three possible changes: remove FALSE values, change the column type, and modify the nullable state. You can see changes in the GIT diff for the DDL. However, applying these changes manually can be tiresome for a large DB, so you can also query the database meta information to automate some of the migration process. For example, a couple of minutes of internet sleuthing led to this for PostgreSQL:

```
SELECT format(
  'UPDATE %I.%I SET %I = NULL WHERE %I=FALSE;',
  table_schema,
  table_name,
  column_name,
  column_name
)
FROM information_schema.columns
WHERE data_type = 'boolean'
  AND table_schema NOT LIKE 'pg_%'
  AND lower(table_schema) <> 'information_schema'
  AND is_updatable = 'YES';
```

Executing this in a query window produced lines like the following as output.

```
UPDATE public.fact_type SET is_derived = NULL WHERE is_derived=FALSE;
```

You should then selectively review these lines (against the DDL differ) and execute the filtered SQL to remove all FALSE values from true-only columns, then use the following to produce additional DDL to modify the column types (you'll see the generated BOOLEAN_TRUE domain when you diff your DDL. Make sure you add this to the database first).

```
SELECT format(
  'ALTER TABLE %I.%I ALTER COLUMN %I SET DATA TYPE public.BOOLEAN_TRUE;',
  table_schema,
  table_name,
  column_name
)
FROM information_schema.columns
WHERE data_type = 'boolean'
  AND table_schema NOT LIKE 'pg_%'
  AND lower(table_schema) <> 'information_schema'
  AND is_updatable = 'YES';
```

To produce executable output like this, which again may need to be filtered based on the DDL diff.

```
ALTER TABLE public.fact_type ALTER COLUMN is_derived SET DATA TYPE
public.BOOLEAN_TRUE;
```

Once these changes are in place your database will be migrated sufficiently to a state matching the current DDL and be ready for continued use.

Default Values

With an available True/False unary pattern producing a non-nullable role, I wanted to make a default value available to simplify initial data population. However, there was no facility for default values in the tool, so I added a default value capability to all roles (except subtype and supertype roles) that map to single columns in the database. The relational mapping for this is not guaranteed because a default value on an absorbed subtype (or similar construct, like an objectified unary fact type) cannot be mapped to the relational model. With absorption, all columns for the subtype must be null, so setting a default value on these columns would force the supertype to be of the subtype for every new insert.

The mapped default values are associated with a role, but the system itself begins with a value type. Setting the *DefaultValue* on a value type simply provides a default for roles of that type, it does not produce relational output for the type. From an implementation perspective, default values are very similar to value constraints, which can automatically apply to downstream roles (for example, a value constraint can apply to an identifier that ultimately uses the value type). So, default values and value constraints go through similar validation—they both must satisfy the data type and any context value constraints.

The difference between value constraints and default values (obviously apart from the purpose of the construct) is that value constraints cannot be arbitrarily turned off for a given role, whereas default values can be. So, the *DefaultValue* property on a role will always have a choice of *<No Default Value>*. If a context default value is available, it will be the default and show *<Context Default Value> = VALUE* in the dropdown. In this case, a special state is saved to remove the context default. For string types, you may also want an empty default value. This empty needs to be distinguished from the context default and is also stored as a special internal state. From the editor perspective, however, simply delete the text in the DefaultValue property to enter the empty default. You can choose *<No Default Value>* (or a context value) to remove this empty default.

The default values are not shown on the diagram, but they are mapped to the relational model (except for absorbed types) and are also shown with the role verbalization. A modified default value will appear in bold in the Properties Window (except for the empty state, where there are no characters to bold).

Other Changes

There are a few other changes and defects fixed in this release. Fixes happen organically when you touch and test over 250 files.

Diagram Spy *All Diagrams*

The one notable change is with the *Diagram Spy* window. This window is meant to show an alphabetized list of diagrams when it is first opened. However, this feature has never worked beyond small models. Basically, if you use a .NET LinkLabel control, the paint routine fails if you have too many labels, so the links were removed if the paint routine failes. [I always figured .NET would fix this, but they never did.] This view no longer uses the LinkLabel control, so you now get clickable diagram lists for all models (not just trivially small ones).

Since the *all diagrams* view is reliably useful now I added a menu item to get back to it. So, from a diagram in the *Diagram Spy* window, you can now choose *Show All Diagrams* from the context menu to get back to the diagrams list. You will also see visited diagrams in a browser 'visited' color until the list is reconstructed (diagram additions/deletions/name changes, or a deactivate/reactivate of the model). The visited colorization conveys useful information, so I did not block this default behavior.

Smaller Relational Changes

As part of the default value analysis, I noticed that downstream value constraint validation had not been implemented. For example, if you have a value type value constraint of [10..20] and a role value constraint using that value type of (9..20] you would not have an out of range error. This was clearly incorrect because the (9..10) portion of the range on the role is not allowed by the value type. To counteract this missed validation, the DDL would produce check clauses for all of the context value constraints, so you would get a check clause to enforce both (9..20] and another (stronger) one for [10..20]. With this validation added to the tool you will only see the nearest value constraint, which is guaranteed to be a subset of earlier constraints if the model errors are clear. This may remove some value constraint clauses from the DDL.

Another annoying issue with the DDL is that large check constraints produced to enforce subtype mandatory constraints on absorbed subtypes would frequently reorder column names when the model changed. In this case, the secondary grouping of the referenced column names was based on the internal (GUID) identifier of the column, not the name, and the ids can regenerate when the model updates. This resulted in equivalent but reordered DDL, producing text diff that needs to be evaluated. This has now been updated to use the column names, which means future changes will be more stable in this area. However, for this initial change, you are likely to see some reordering in the generated DDL produced by this new NORMA version.

Sample Population

The sample population was updated to use a checkbox for a unary state. Populating the negation will require the negation unary to be selected, either in the model browser or by dragging out a shape.

There was also a major hang resulting from recent auto-generated identifier changes (part of the addition of the UUID data type). This added the *<Auto Generate>* tag to the sample population drop down. To handle cases where an object identifier used an indirect identifier (a subtype, another type that used the auto-id type as part of its identifier, etc.) we recursed the identifier structure to determine if the *auto generate* entry was needed. This recursion code had a typo (*pid.RoleCollection* instead of *recursePid.RoleCollection*) that caused an infinite loop.