Metathesaurus Editing: NCI-NLM Editing Systems Workgroup

Contents of this Page

STATUS After being put on hold, project has been rebooted to research new upgrade options and potential for machine-learning application. 3/30/2016.

Goals of this Task

To create a business case document and high level requirements that can help determine next steps forward as appropriate with respect to any changes to current Metathesarus editing/ QA/ maintenance environment or development of new system or components.

NCI-NLM Editing Systems - Supplemental Documents

This section contains draft documents, meeting notes, and other supplemental materials. New versions of the Business Case and Requirements document will be developed and posted at the end of the business week and/or after major changes from the content below, for the purpose of version control.

- NCI-NLM Editing Systems Launch Meeting Notes
- MEME/Jekyll Demo 1 Notes - Editing systems
- MEME/Jekyll Demo 2 Notes - QA systems
- MEME Outline draft 1.5
- MEME Outline draft 8
- MEME Outline draft 9 - posted 6/22/2015
- MEME Outline draft 10 - posted 6/25/2015
- MEME Outline draft 11 - posted 7/2/2015
- **MEME Outline draft 12 - posted 3/30/2016**
- Notes from MEME Working Group Session 7JUL15

Potential Future of Metathesaurus Enhancement and Maintenance Environment (MEME) Software

Business Case and Requirements Outline

1. Executive Summary

The Metathesaurus Editing and Maintenance Environment (MEME) is a software system used by NLM and NCI to edit terminology content to be exported for production of NLM's Unified Medical Language System (UMLS) and NCI's Metathesaurus.

A number of conditions are driving NCI and NLM to jointly undertake a business analysis of MEME and the UMLS/ NCI Metathesaurus production software and workflow components.

1. Components of the software have been modified over the years, but the software is aging; the Jekyll editing interface is the oldest component, at about 15 years old.
2. The tooling is extremely specialized. Not many people know how to make the whole process work from one end of the production process to the other. It functions well at the moment, but maintainability may be an issue in the future.
3. Reduced (long term) Costs - NCI and NLM have varying business needs, but are running systems in parallel which duplicates systems costs and editing costs. Sharing some of the editing at minimum, could reduce long term costs. Investigating ways to share servers or databases could reduce costs further. And improving production efficiency (see 6 below) and reconciling the domain models (see 7 below) would also reduce cost in the long term.
4. Editing Efficiency – The current system and processes are really inefficient for users. Improved editor productivity should decrease costs.
5. Improved Quality – The tooling could be developed with the aim of improving editor consistency/accuracy. The ability to make the entire editing process more systematic, deterministic, and focused would improve both efficiency and quality. Systematic means that editors are given specific tasks targeted to events within the system such as source insertion, quality assurance, and sampling; deterministic means that work is generated in reproducible and reliable ways (e.g. built around STYs, built around sources, built around QA), and reproducible means that "same kinds of editing" produce the same results.
6. Production Efficiency - In the current system, in order to publish the UMLS Meta, we must
   - Turn off editing for a day or two
   - run pre-production
   - produce and QA RRF files,
   - produce/QA a DVD image that we eventually distribute for our users.
   - Load the RRF files into our web services (UTS API/browsers).

   The entire process takes around 4 weeks. While we've made (great) improvements to this process over the years, we are still a bit hamstrung as we must wait until 2 systems (MEME/MMSYS) can reconcile their respective RRF outputs before we can release the data publicly and update our APIs/browsers.
7. Reconciling Domain models - Right now MEME, RRF, and UTS all speak a slightly different language to represent the same types of things. The result is that we have processes and data transformations (like loading RRF into UTS) that are prone to errors and require a lot of manual hand-holding and additional QA.

This wiki page documents the effort to understand the needs and requirements for editing the Metathesaurus content, with a goal of identifying possibilities for how NLM and NCI might collaborate to upgrade or rebuild the editing software in the future.

In order to address some of the goals/drivers listed above, future approaches identified should look for ways to streamline and improve the editing and production processes for NCI and NLM, for example, by:

1. Upgrading the software base using modern software production practices as feasible, like continuous integration, planning for providing some early benefit before all pieces are complete.
2. Identifying a way to invert shared sources once.
3. Identifying where NCI and NLM business rules for editing can be harmonized.
4. Reducing maintenance costs through the possibility of running 1 set of servers.
5. Considering whether there is a technical approach for running 1 database that can meet both sets of needs by enabling creation of separate views, or 2 databases with ability to synchronize specific content on demand.
6. Providing a web based editor interface and more integrated tooling to promote efficiency.
7. Improve the publishing efficiency by creating a more flexible, web-first publishing system.
8. Reconciling the different domain models in use in different parts of the editing and publication processes. Perhaps this can lead to a smoother approach to publishing.

Assumptions: Most of the documentation below applies mainly to the central editing portion of the UMLS and NCI Meta production processes. The early and late stages of the Metathesuarus workflows most likely are not going to change initially– i.e. how source terminologies are processed for insertion and how the exported files are packaged and served for production. Note that we are NOT assuming a priori that the editing and workflow need to be exactly replicated, though we have tried to understand what basic features any editing system would need.

2. Overview of MEME

The purpose of the Unified Medical Language System is to assist in the advance of systems that aid health care professionals and researchers in retrieval and integration of biomedical information obtained from a number of knowledge sources, and to help users in linking dissimilar information systems. UMLS also builds knowledge sources that can be used by intelligent programs to overcome:

- Language disparities used by different users or different information sources.
- Disparities in granularity and perspective.
- Problems in mapping and aggregating data from different sources.

MEME contains information regarding biomedical concepts and terms from many controlled vocabularies and classifications used in patient records, administrative health data, and full text databases. The Metathesaurus maintains the names, meanings, hierarchical contexts, attributes, and inter-term relationships present in its source vocabularies. General defining principles of the Metathesaurus include:

- Concept Organization
- Common format for distribution of vocabularies
- Linking vocabularies
- Representing multiple hierarchies
- Regular updates (at least annual)
- Representation of the meaning in each source vocabulary
- Explicit tagging of each source vocabulary's information

The Metathesaurus is produced by automated processing of machine-readable versions of the source vocabularies. To accomplish this, each source vocabulary must be put into a common representation. After the source is in this common data structure, lexical matching is performed, suggesting possible synonyms and concepts to which the added concept might be related.

3. Mission of this Activity

The mission of this activity is to understand the general requirements for the Metathesaurus editing system, as illustrated by the MEME environment (Figure 1 below). The conclusions from this task may allow for updating of the Editing System within MEME, and then, if possible, stage a long-term plan for "re-doing" or updating multiple components of the system. Both will be addressed in the following sections. The goal in the near term is to understand what features and functionality would be required of any Metathesaurus editing system.

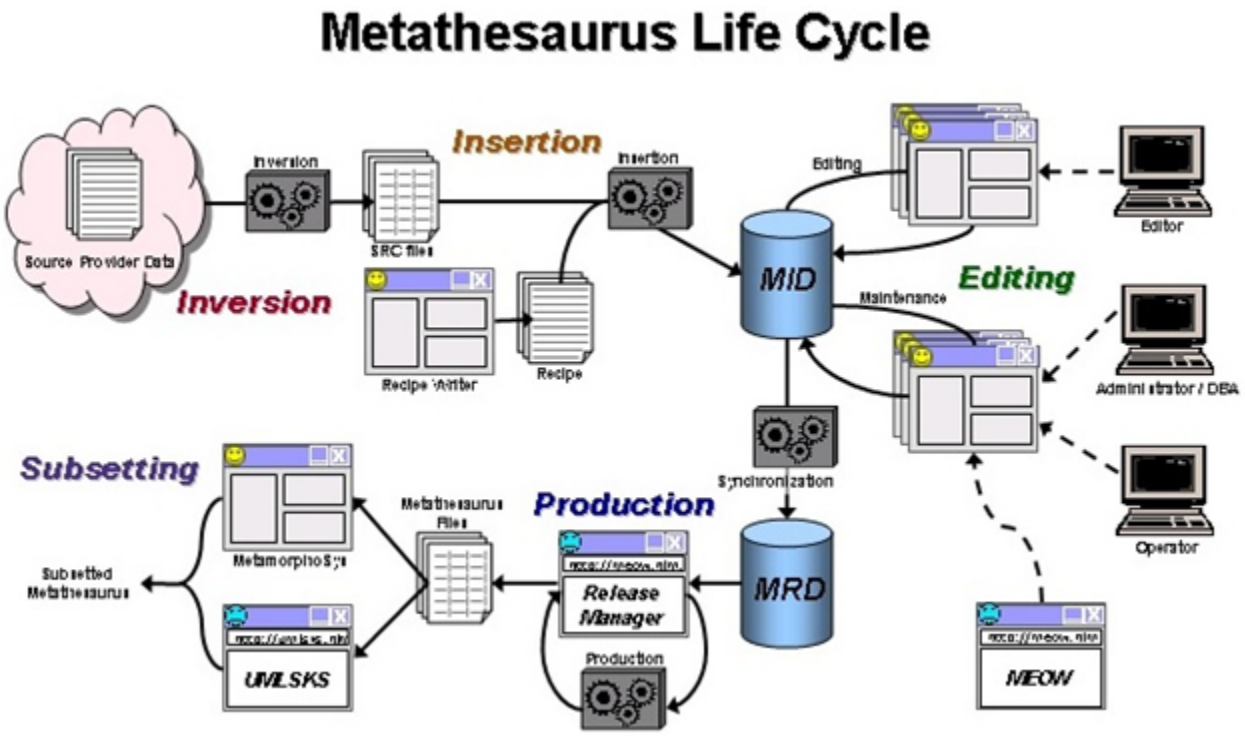


Figure 1 – Metathesaurus Lifecycle

4. Purpose and Scope

The intent for understanding the editing requirements is the main focus, but high level requirements of other phases of the lifecycle are important to understand in context, as they interact with the editing system, as seen in Figure 1 above. The intent is not necessarily to replicate MEME, but to derive requirements based on observation of these tools and their use.

5. Stakeholders

A. Content Editors

B. Workflow managers

C. UMLS and NCI Meta production staff for all phases of life cycle

D. NLM management staff

E. NCI management staff

F. Software Developers

G. End users

6. Roles

The following roles are current roles in the MEME environment. These roles currently have some overlap in responsibilities and some are rarely used, therefore one role does not necessarily represent one individual. Any possible update or upgrade of MEME will have to reassess the Roles for the system.

A. **Administrator (R-A):** Administers the IT infrastructure, hardware, software installations, and databases. Responsible for configuring new systems, installing software, scheduling recurring processes, and making copies of databases as needed. There are at least three sub-administrators

a. Database Administrator (R-A.dba) - Database

b. System Administrator (R-A.sys) - Operating System

c. MEME Administrator (R-A.meme) - MRD, MEME, MIDSVCS, LVG, other servers

B. **Developer (R-DEV):** Maintenance of existing code line development of enhancements based on objectives/priorities. Includes management of the complete software development cycle, including software QA.

C. **Development Manager (R-DM):** Manages the software development life cycle (including documentation and QA).

D. **Documenter (R-DOC):** Maintenance of existing documentation, development of new ways of tracking processes, reporting progress, etc.

E. **Editor (R-E):** Edits MID content.

F. **Editing Manager (R-EM):** Manages the editing process. This includes deciding when insertions will happen, and performing the routine EMS /WMS operations after relevant DB work is completed.

G. **Inversion/Insertion Manager (R-IM):** Manages the inversion and insertion process. This includes prioritizing inverter resources and making sure deadlines for source insertion objectives are met. Also plans and manages bug fixes and development of enhancements and QA improvements to inversion toolkit. Plans staging of test and real insertions to meet production timing goals.

H. **Inverter (R-IV):** Writes inversion scripts, performs inversion work, and validates results. Typically also responsible for things like converting to UTF-8 character set and providing the gold standard "initial state" for each source.

I. **Insertion Operator (R-IO):** Takes inverter output, develops insertion recipes, and performs test and "real" insertions.

J. **Maintainer (R-M):** Performs ad-hoc maintenance operations. This can be in response to QA problems, or requests for E or EM roles for specific work (e.g. change all 'XX' STYs to 'YY' STYs for concepts with source S1 where they have X type of relationship to S2).

K. **Production Operator (R-PO):** Performs pre-production operations, production operations, and post-production operations.

L. **Project Manager (R-PM):** Overall project manager.

M. **QA Operator (R-QA):** Operates QA tools. Identifies QA problems at each stage of life-cycle and triages for handling. Communicates with M role to determine resolution of known problems. Also responsible for QA reporting functions.

N. **Source Liaison (R-SL):** Responsible for performing source acquisition and initial qualification and evaluation.

O. **User Community Liaison (R-UCL):** Responsible for interacting with user community. Triage of user complaints and suggestions and routing of requests or questions to relevant roles.

7. Overview of Editing Cycle and High-Level Requirements for MEME

This section outlines the high-level requirements for MEME and gives a snapshot of where the system is now. The following sections, starting at I. and following beyond, outline the evolution of MEME, while focusing mainly on the Editing functionality.

A. Inversion

Inversion is the process of converting source provider files in their native format to MEME's common format, called the .src format. Generally, the inversion process consists of running a script or scripts tailored to the particular conversion task.

Inversion begins with a set of files that come from a source provider (for example, MeSH). These files will be in a source specific format. One key goal of inversion is to maintain source transparency, or the preservation of all initial data in the new format. Source transparency can be validated by attempting to recreate the original source files. Additional work is required for sources that have hierarchies to generate the necessary information to completely and accurately represent those hierarchies.

Inversion ends with insertion-ready .src files and a source insertion recipe used to actually load the inversion data into an editing database.

Information about source data and the process of converting it is stored and maintained over time using the Source Information Management System, or SIMS. This is an on-line documentation system that captures all relevant information about the process and data. Eventually, a subset of this data is to be made publicly available to satisfy another requirement of source transparency, that the representation of original source data in MEME is accurately documented.

A hierarchy viewer tool supports visualizing and sampling source hierarchies prior to insertion. This is a good check on whether hierarchies were built properly.

There are several strong justifications for continued use of a common load format for the system:

- Insertion recipes can be written against a reference data model that does not change, facilitating ease of understanding and code reuse.
- Quality assurance processes can perform structural and semantic validation in a consistent way. Similarly, accounting for changes in source terminologies from version to version can be done consistently, regardless of changes in underlying native formats.
- Counting and sampling approaches to data quality can be applied consistently across the entire life cycle. This is particularly valuable for ensuring "conservation of mass".

Current .src is a historical format and some consideration should be given to long-term harmonization of the input and output formats for the Metathesaurus (e.g. ".src" and ".RRF"). Converging on a single, standard data representation will enable "closing the loop" allowing QA processes to verify that exactly what goes in is exactly what comes out.

B. Insertion

Insertion is the process of loading inversion data files into the Metathesaurus Information Database (MID). The source insertion process typically begins with inversion files. A GUI wizard called the “recipe writer” is used to create a “recipe”. The recipe informs the insertion system of the series of steps needed to load and merge the inversion data with the editing MID. Behind each step is a piece of code that performs some aspect of the insertion. When finished writing a recipe, a user creates an HTML file for viewing the recipe steps, an XML document representing the saved form of the recipe, and a UNIX shell script for performing the insertion.

When inserting a source, a test insertion must first be scheduled and run. The source inverter and NLM content experts should review the test insertion to make sure that (a) the insertion recipe was properly implemented, and (b) that the insertion recipe was the correct recipe for the data. If any major modifications are made, an additional test insertion must be performed.

A key requirement of insertion process is that the underlying data store must be able to accommodate multiple, simultaneous versions of the same terminology. For efficiency, it is even more desirable to be able to represent a shared core of data between two versions, the delta of the old version (changed and removed), and the delta of the new version (changed and added).

C. Editing

Editing is the process of resolving conflicts between a source's view of synonymy and the UMLS view of synonymy. The Metathesaurus is a thesaurus comprised of a large number of constituent vocabularies, and it is an editor's task to incorporate all of the different views of the world into a homogenous single representation.

Editing is primarily driven by four processes:

A. Insertion of new sources. A new source is one whose content has never appeared in the Metathesaurus before. Here, the insertion process attempts preliminary matches to the existing MID content. It is ultimately the editor’s task to review every concept in the new source and decide if the insertion algorithm placed it correctly into a UMLS concept or if it should be moved elsewhere. Typically, editors try to merge as much of the new source with existing MID content as possible so as to avoid creating new missed synonymy. Another important goal is to locate all new concepts created by the source and connect them to existing MID concepts so as to avoid creating new orphans.

B. Insertion of updates to existing sources. An update source is a new version of a source that already exists in the Metathesaurus. Here, the primary editing responsibility is the same as for a new source, but instead of reviewing all content, editors do not have to review content judged to be safe replacement.

C. Quality assurance predicates (QA bins). Various scripts and PL/SQL packages produce sets of concepts (clustered or non-clustered) that violate certain QA conditions.

D. Ad hoc sets. At any time, a certain predicate may be used to generate a list of concepts. Frequently used ad-hoc sets are incorporated into a collection of Ad Hoc Bins.

Editing management today typically revolves around bins, or collections of concepts. Bins are managed and maintained by the Editing Management System. This web based tool allows administrators to create mutually exclusive bins of concepts based on an ordered list of predicates. Additionally, orthogonal sets of bins based on QA queries or just frequently used ad hoc queries can also be built and maintained. Editors are assigned worklists which are collections of concepts from a particular bin. Worklists are created and managed by the Worklist Management System.

An editor typically looks at one or more UMLS concepts at a time and attempts to resolve issues that cause those concepts to require review. Once an editor decides that a change needs to be made, they will use the MEME4 editor (Jekyll) to enact the change. An editor is required to approve the concept when finished with it. The concepts on an editor's worklist that are left untouched are subjected to a process called “stamping,” in which they are algorithmically approved.

D. Production

Production occurs post-editing and is not the main focus of this requirements activity. (However, it should be considered in the context of the longer term plan to change UMLS Editing, QA and Production.)

The steps and elements are as follows:

a. Synchronization

b. Metathesaurus Release Database

c. Release Manager (Production)

1. Quality Assurance- QA on MEME published reports to review and reconcile differences.

d. Metathesaurus Files (mmsys.zip and .nlm files)

e. DVD Mastering

f. UMLSKS

1. User Applications

2. UMLSKS User

h. MetmorphoSys- Further information is available here.

Some consideration should be given to what would constitute a "continuous release" process for content. The present system requires a "freeze" of editing in order to validate and prepare content for a release. Following those steps, editing is allowed to resume in parallel with the generation of release data. The main alternative to this approach would involve maintaining a "release ready" state of the data where editing is done in a branch or separate environment and then final approved content would be merged in to create a new "release ready" state. Such an approach would facilitate "anytime release" as well as the ability to produce a delta release to provide users only the content changes that have occurred since the latest release.

Web-first publishing might streamline the production process. (Add more detail from Steve)

E. Other overall considerations:

Reconciling Domain models - Right now MEME, RRF, and UTS all speak a slightly different language to represent the same types of things. The result is that we have processes and data transformations (like loading RRF into UTS) that are prone to errors and require a lot of manual hand-holding and additional QA. Is it possible to reconcile the domain models in the short or longer term?

8. New Editing System Functional Requirements

These functional requirements are derived through a series of meetings, demonstrations, and review of documentation.

A. General Editing System Requirements

a. [All functionality on Jekyll's current SEL (Select) screen] SEE SCREENSHOT.

b. Ability to edit:

1. Semantic Type

i. Add, Delete, Replace functionality

2. Concept

i. Merge, Split, Approve functionality

3. Atom

i. Add, Delete, Move (from one concept to another) functionality

4. Relationship

i. Change, Add, Delete functionality

c. Lifecycle tracking of individual source vocabulary and where individual sources are in the lifecycle (see Figure 1)

d. Application should allow for:

1. Browsing in Editing mode

i. All browsing functions need 'back', 'forward', 'go to beginning', 'go to end functions' while browsing Editing system

2. Browsing a concept

i. Ability to view multiple concepts at the same time/within same window.

ii. Once insertion is done, need to be able to see relationships, hierarchy, etc within database without looking it up in a book. See for example, RX-NORM editor where you can visualize the source well.

3. Browsing published concepts

4. Browsing of a concept within the full lifecycle

5. Multiple (e.g. max 15) users to edit simultaneously (and see each others work?)

6. Role based functionality (e.g. workflow manager, editing manager, editor)

7. Reporting

    i. Parent/child hierarchy in reports (export to file for download?)

    ii. Ability to View All or Filter

8. (configurable) Real-time integrity checking to prevent bad data input.

9. Export (separate from release?)

e. Interface with the worklist environment (e.g. discover assigned worklists, pull concepts into environment from worklist, track progress through worklist, stamp worklist when finished – or mark for later stamping)

f. Functionality that supports publishing at intervals in XML and OWL (publishing what? Metathesaurus itself is NOT conducive to Owl as it is not an ontology)

g. Sub-setting capability.

B. Worklist Functionality

Worklist integration should be simple and efficient, so that editors can utilize quickly and painlessly.

a. Ability by workflow manager role to view worklists in a dashboard within the web-based system

b. Ability to assign tasks via worklists.

1. Worklists may be as small as a single concept

2. Worklists may contain concepts or atoms from one or more sources

3. Worklists support clustering (e.g. groups of concepts connected, say because of a demotion)

c. Tracking of edits and assignments via worklist (delta reports indicating what has changed and who made changes)

C. Editing (by Editor)

a. Tool should help editors assert synonymy

b. Tool should allow for sort, flag, and move functions

c. Ability to view and edit whole cluster (group of concepts) and/or each concept one at a time

d. Ability to split/merge concepts

e. Ability to see relationships and hierarchy within database

f. Ability to Undo previous editing steps

g. Multiple simultaneous users without conflict (or concurrency issues, like deadlocking)?

h. Edit semantic type, flag for status

D. Bins (or equivalent functionality for holding sets of concepts for review/ editing meeting specific criteria)

There are currently three kinds of bins used to support editing: ME bins, QA bins, and AH bins.

Mutually Exclusive (ME) are the major bins, primarily used to drive editing and are *based on concept status as well as other predefined conditions.*

A new system needs an equivalent functionality, with ME bins holding *concepts from multiple sources*.

Quality Assurance (QA) bins hold concepts *based on QA rules that can be run at any time* to identify known, problematic data conditions. Typically work from these is put into “checklists” and edited alongside the worklists. If something is already on a worklist, it typically does not also get put onto a checklist. QA bins should also be at “zero” for the system to be ready for a release. Similar functionality is needed for a new system.

Ad hoc (AH) bins are a less commonly used mechanism for weaker QA rules but it has a history mechanism. The idea is that you can find concepts matching a condition where sometimes that condition is acceptable and sometimes not. Editors make choices about that and the *history mechanism prevents these cases from needing re-review*. QA bins are more for rules that should not be violated, whereas AH bins are for rules that have known and correct exceptions.

Overall requirements for Bins or equivalent:

- Bins are configurable.
- Bins can be created and reordered.
- Bins are the same if the same editing rules are used, and different bins used to support different editing rules.
- “Comment out” bins that you don’t want at particular time.
- All bins should have a zero count to freeze for production
- “ncibadmerges” – This is an NCI specific bin used when two NCI/PTs get merged into the same concept. The fix is usually to separate them back out along with any other atoms that mean the same thing (e.g. split the case). Occasionally, these are correct and then the information is fed back to the NCI thesaurus authoring environment to actually put them together into one code.
- Q/A goal : ‘it would be nice to be able to make bins without going to the programmers’

E. Quality Assurance (QA)

QA of the types mentioned below, are now controlled outside of the Jekyll editing interface, through a series of tools and scripts which act on the .src files directly.

a. Ability to QA via ‘conservation of mass’- Counting, comparing, and explaining discrepancies. There is a “qa reference model” of the data which determines the aggregations (group by) expressions that are used for counts. Counting is all done relative to the “release perspective”. An adjustment/explanation mechanism should also be supported to allow for known discrepancies.

1. A count-comparison function is also needed to verify that mass is preserved across the life cycle (and that known explanations account for all discrepancies).
2. Should be able to compare counts of a source to the previous version
3. Should be able to compare counts of the current release to the previous release
4. Should be able to compare counts within editing environment from week-to-week.
5. Should be able to compare counts from before a life-cycle event (like an insertion) to counts afterwards. – e.g. assure that an insertion actually brought all expected source data into the system.

b. Hierarchy Viewer function to ‘spot check’ QA from Insertion

1. Hierarchy visualization should be based around data that is actually used to insert into the database. Current tool builds an intermediate form that is used by the viewer – this is problematic because it may be out-of-sync with the actual insertion files.

c. Source (SRC) QA: For Spot Checking purposes from Insertion- Automated QA algorithm runs every time a source is inserted into the system. When things inserted, able to go in and view data elements to make sure that everything got inserted as it should have. In particular, it is important to sample new content (e.g. a new term type) and content that has been removed since the previous release.

d. RAW SRC Count: SRC QA run data for spot checking Inversion- Whenever inverters run an inversion, one script is a SRC QA to show breakdown. First time things are counted and compared, if something is different, then it must be explained. Ability to labelling sequential order in which stages happen, e.g. “presentation improvements”.

e. Other QA tools??

1. Rule-based checking of data conditions not completely managed by database constraints (e.g. the “MID Validation” tool).
2. Statistical analysis to look for outlier conditions (e.g. very low volume combinations of semantic type usage) – this is good stuff for “ad hoc” bins because of the history mechanism.

f. Requirement related to Editing: QA tools must feed into Editing System to create a checklist that is assignable to editors internally.

F. Editing system must be a web-based application

a. Eliminate extra layers and/or “doors” into system

1. Citrix
2. Multiple logins
3. Remote Desktop/Applications (NCI)

b. User must be able to login using NIH credentials.

c. Simultaneous Editing with multiple users logged in

G. Collaboration with NCI (and other organizations)

One of the goals of this activity is to find ways to reduce effort and cost involved in editing and producing NCI Meta and UMLS, and also to facilitate collaboration (and editing with and by other outside organizations as appropriate). To that end, there are a number of questions to be investigated from technical and management perspectives.

a. Is there a way to invert shared sources once?

b. Is it feasible to have a single server that can be used by multiple organizations? Or independent servers that can be integrated and/or pass information back and forth? May be difficult to manage a shared resource but there is precedence for inter IC agreements of this sort. Software should be modular to not affect editing interface as pieces underneath are developed, switched out.

c. Would a 'publish-first' mechanism facilitate b above?

d. If there were to be a single server, could different user edit concepts differently (i.e. disagree on synonymy)?

e. Can we then export different “versions” of a Metathesaurus?

f. How would inversion/insertion be handled?

g. How could the underlying models for MEME, RRF and UTS be reconciled?

9. Non-Functional Requirements

A. Usability

a. All tools must support publishing at intervals in XML, OWL, etc., with a sub-setting capability

b. Backup on continuous basis

c. Ability to print (through the web browser)

d. Track actions of users

B. Constraints and Dependencies

a. Wherever there is a data transformation step, there must be a corresponding QA step to validate the transformation and identify any missing, extra, or unexpectedly changed data.

C. Extensibility

a. Compatibility if customer is updating systems (‘make it easier for customers to get new content’)

D. Performance

a. Good editing performance with X number of simultaneous editors

E. Technical Requirements- NIH, CBIIT

a. Must meet NIH ISSO Security requirements

b. External API Integration

c. External facing applications must meet 508 compliance

d. Use standard software and tech stack, open source if possible

e. etc.

10. Proposed Plan for Editing System (Phased Approach)

Phase 1 - Editing Functions transition to Web-based Application using existing and new requirements mentioned in previous sections:

A. Web-based Editing Application

a. Workflow

b. Editing

1. Multiple simultaneous usage

2. Ability to merge deleted concepts on server side

3. etc.

c. Quality Assurance

1. Counting, comparing, and QA at each transformation step

2. QA Tools feed into Editing Management System to create checklists/assign work within tool

After several attempts to upgrade the MEME environment in the past, it was identified that the main ‘roadblock’ is the complexity of lifecycle interactions between various processes and modules (See Figure 1 above). After collaborating with MEME users and editors, the overall recommendation is to create interfaces (APIs) or standard ‘handoff’ artifacts that separate lifecycle stages in ways that completely disentangle them. By creating well-defined file formats and REST APIs, the core functionality and the interface tooling of each lifecycle stage can be cleanly separated from one other. This allows for incremental reengineering of individual pieces; i.e. Editing System, then Inversion, Insertion, Publishing System, etc.

Each of these systems is currently implemented with legacy technology and can be upgraded without requiring an upgrade of the entire system. The starting point would be to upgrade the interfaces between lifecycle stages, thus allowing tools and interfaces to be upgraded without first requiring the back-end to be upgraded.

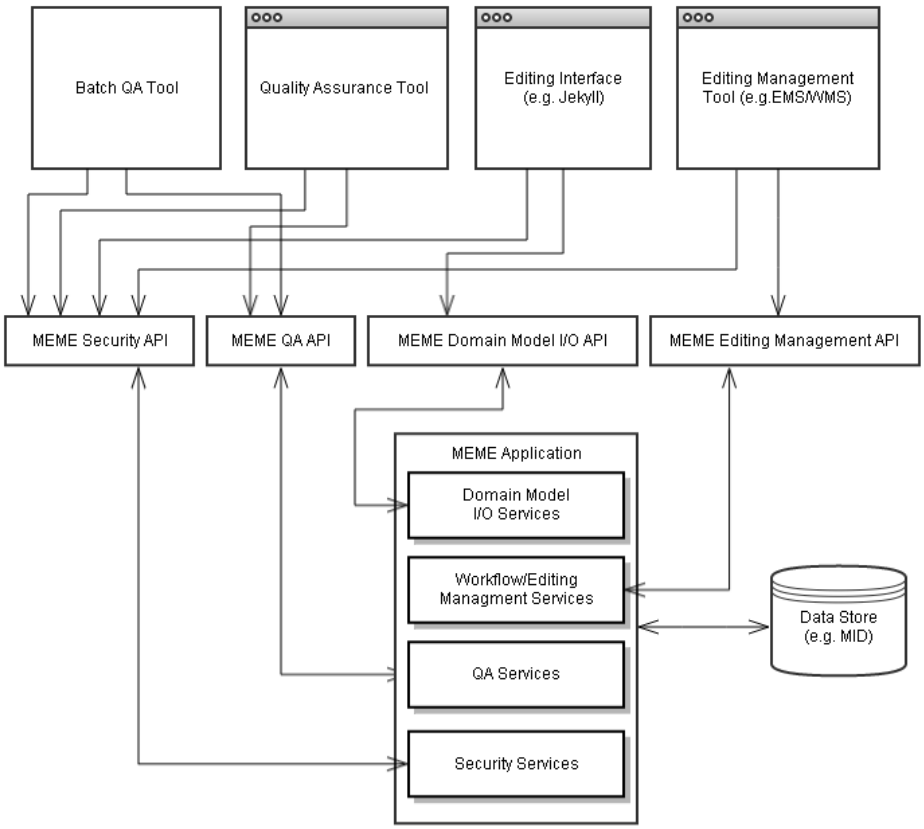
Stage 1: Define security and domain model Read/Write REST APIs. Implement them to run against the existing MEME environment.

Stage 2: Redevelop the editing interface (currently Java application “Jekyll”).

Stage 3: Redevelop other tooling that would make use of the corresponding API (in order of priority)

Stage 4: Upgrade backend to directly implement the APIs using modern architecture and development approaches.

Consider the following diagram that shows the fully reengineered MEME Editing System (Figure 2). There is an API layer between the backend system and the tools and interface that drive the lifecycle processes.



MEME Architecture - Editing and Editing Management.gliffy

Figure 2 – Editing and Editing Management System

The best incremental pathway to a complete system would be to define and implement the middle layer APIs. For example, with ‘Security Services’ and ‘Domain Model I/O’ APIs written against the legacy system, but using a next-generation set of domain model objects and services, would enable the complete redevelopment of the editing interface without needing to change any other feature of the system.

Similarly, between each ‘major functional component’ of the system there is a ‘bridge component’ that allows either side of the bridge to be reengineered without changing the other side. The long-term implication of this is that each major functional component ends up independent of one other, therefore any future needs for redevelopment can also be incrementally undertaken. Major functional components and bridge components are defined in Appendix A.

11. Proposed Plan for Remaining Components (Phased Approach)

Phase 2 - Transition of other parts of Metathesaurus Lifecycle to updated System in similar fashion to Phase 1 for Editing System. Figure 3 is for Inversion and Insertion and Figure 4 is for Release and Production.

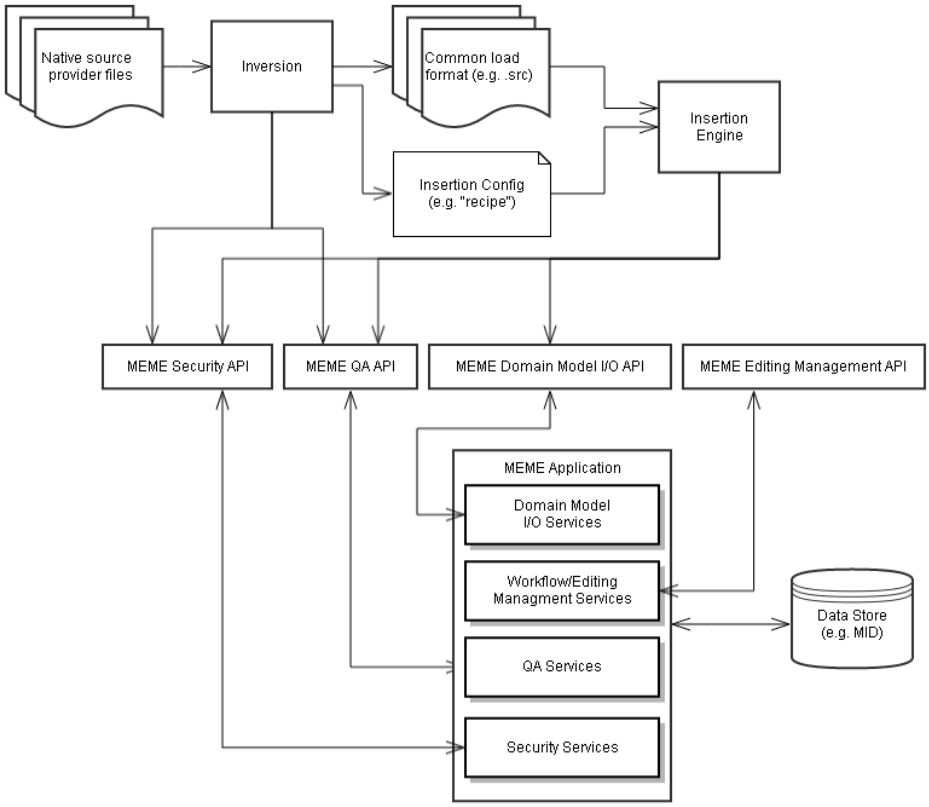


Figure 3 Inversion and Insertion

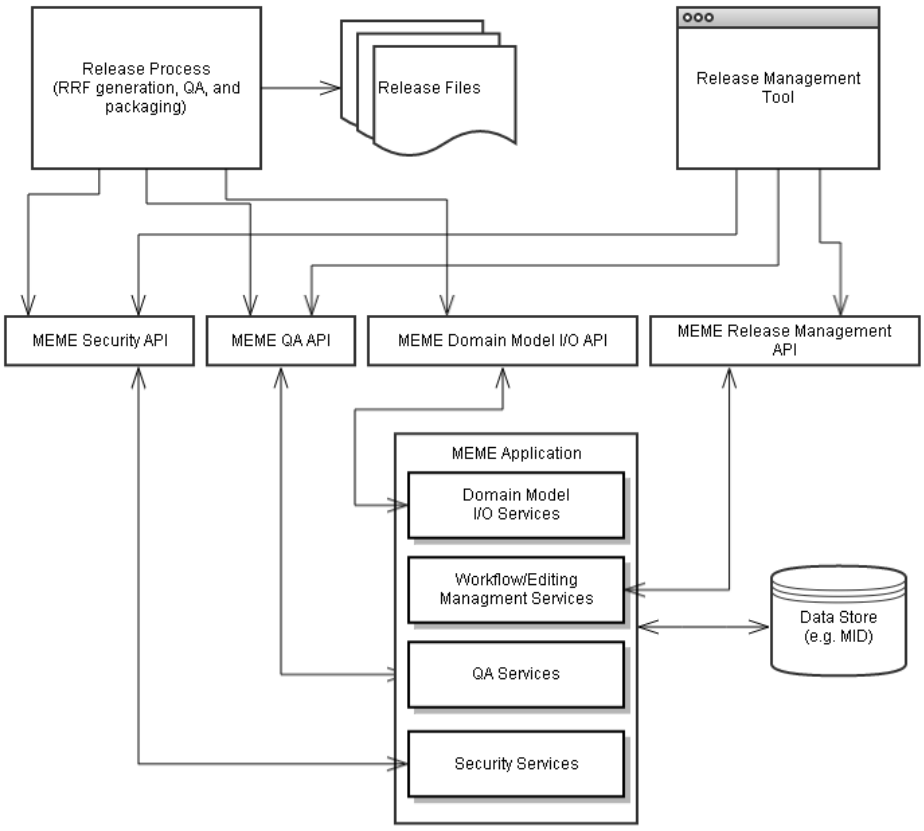


Figure 4 – Release and Production

12. Alternative Upgrade/Research Project

Option 3A- Applying web-based terminology server/browser that already exists but is not being implemented

West Coast Informatics has developed a lightweight terminology server/browser platform that can host individual terminologies, or a Metathesaurus itself. In the case of NCI-NLM collaboration, it can also host multiple Metathesauri reusing the same underlying terminology versions. It can also host multiple versions of the same terminology at the same time (thus handling terminology updates, etc).

The basic fields and features to support editing (e.g. identifier assignment, etc) exist in the back-end but are not exposed at all. We continue to develop the platform (recently adding relationship search capabilities and advanced searching). We are doing an infrastructure overhaul of the UI to support things like URLs directly loading a particular concept, and the ability to view multiple concepts in different windows within the same application framework. That will form the basis of the next step which is to develop basic editing features. Approaches for user roles would be reused (e.g. ADMIN, REVIEWER, AUTHOR, VIEWER) and approaches to workflow (SIMPLE, SINGLE REVIEW, DUAL INDEPENDENT REVIEW WITH CONFLICT RESOLUTION).

The option 3 would be to develop this platform out further to support:

- Role-based access
- Metathesaurus editing actions (MERGE, MOVE, SPLIT, STY change, etc)
- Workflow support (workflow paths, “available work” grouped by criteria, and “assigned work”)
- Insertion capabilities (new versions of terminologies – reuse of what is “in common”)
- Built around “.src” model so that inversion can remain the same.
- Tweaks to support the stranger aspects of Metathesaurus (e.g. handling MSH, dealing with the “SRC” problem, etc).
- Release process to produce RRF – the model is essentially built around RRF.
- Initial quality assurance hooks

Option 3B- Machine learning research project

After 3A is implemented, tested, and approved, machine learning algorithms would be introduced into the process to facilitate editing. Determine where editing can be done algorithmically and determine where information was needed from users for “supervised training”. Machine algorithms could also drive QA and identify cases of things that look wrong (e.g. weird STY combinations, or STY combinations in conjunction with hierarchies that don’t make sense, etc).

13. Management of Chosen Approach Implementation

A. Work Breakdown Structure

B. Schedule

C. Funding/Budgetary Proposal

D. Organizational Constraints/Considerations

E. Risk and Risk Mitigation

F. Conclusions

G. References

1. UMLS Metathesaurus Editing Manual. https://meme.nci.nih.gov/Manual/Complete_Training_Manual.pdf

2. Metathesaurus Lifecyle. https://meme.nci.nih.gov/MIDS/.

3. Carlsen, Brian. ‘Migration Plan Ideas’ and figures 2-4.

Appendix A

The major functional components are as follows:

- Inversion Processes (currently implemented as shell/perl scripts)
- Insertion Processing (currently implemented as a shell script generated by a Java recipe writer. Ideally, enough information about the “recipe” could be defined at inversion time that a simple engine could process a reusable configuration artifact and automatically enact the insertion.
- Batch QA Processing (background analysis of various states of the data)
- Editing interface (currently Jekyll)
- Editing/Workflow management tooling (currently EMS/WMS)
- Metadata management tooling (not shown but would be in Figure 3 alongside editing interface, this is all the stuff to manage source data, TTY lists, etc. This envisions that either the “MEME domain model i/o API” supports these capabilities or there is an additional metadata service)
- Admin tooling (also not shown, but would be in Figure 2)
- QA tooling (currently a collection of a wide variety of different, but similar tools, and EMS QA/AH bins)
- Release processing (currently a collection of PL/SQL procedures, java classes, and shell scripts)
- Release Management tooling (currently the “release manager”)

The major bridge components are as follows:

- .SRC files (the abstracted layer between Inversion/Insertion)
- Insertion recipe configuration (also part of the abstracted layer between Inversion/Insertion)
- MEME Security API – for controlling access to the back-end server
- MEME QA API – for reading/writing/comparing sets of QA counts or samples and for recording explanations for discrepancies. (would be used across all lifecycles)
- MEME Domain Model I/O API – for reading/writing domain model and metadata objects, accessing the history of domain model objects, validating, and querying domain model objects
- MEME Editing Management API – for managing “bins”, assigning and tracking work, “stamping”, etc.
- MEME Release Management API – for invoking release processes, monitoring progress of release processes, etc.