ManipuLogic Project Plan

Table of Contents

Introduction

This document will serve as the first revision/iteration of a project plan outline for the ManipuLogic project in GitHub, run by Matt Gracz. It will contain project goals, initial backlog content and priorities, and a project strategy.

Project Goals

1. Create a computer program that allows users to explore formal logic in an interactive and dynamic way, such as:
    a. Simple manipulation of symbols to determine if user-defined formulae are well-formed
    b. Analyze equivalence-relations and other formal relations between two or more formulae, etc…
    c. Postulate a theorem and determine if it can be derived from a user-defined set of axioms.
    d. From a user-defined set of axioms, derive and display a set of theorems within user-defined parameters such as raw-time-to-derive, “levels deep” in a proof graph, complexity metrics, etc...
    e. Other useful or common tasks performed in formal logic that can be automated with a computer program, i.e., that don’t require human creativity and spontaneity.
2. The software should have an intuitive, user-friendly interface, that allows for quick and easy manipulation of input parameters, reader and print friendly output analysis, and the appropriate amount of customization for the user tasks supported by the software.
3. The software should be of a high quality, satisfying non-functional requirements such as:
    a. Fast in terms of computation speed
    b. Reliable in terms of feature defect quality and quantity
    c. Consistent in terms of computational repeatability (e.g., according to accuracy and precision).
    d. Interoperable with e.g., documentation processing software of ManipuLogic’s output, as appropriate.

Backlog Item Definition

Output from backlog brainstorming session:

initial BL items:

arch:

-research better program structure for release candidates

-do we want to allow multiple instances of ManipuLogic to run simultaneously? Currently blocked

-update architectural documentation and create documentation strategy

-define and document initial CLI approach

-define and document initial GUI approach

-choose GUI toolkit, at least for initial development.

-figure out unit testing framework --> use pre-existing or write my own?

-define initial branching strategy: DEV as master. <features> for development and testing, demos for end-of-sprint demos if needed, REL<release #> for release candidates.

-define sprint length and figure out how to track in GitHub (2 weeks, use milestoens?) Make release candidates how often: every 4 sprints?

-feature/story A.C.: feature dev'd, unit tests pass, demo-able, and impacted documents updated.

features:

-Make DEV branch's functionality match the prototype's, at a minimum, so we have a good starting point.

-binary operator distribution

-double negation

-order of operations when parens aren't specified (nice-to-have?)

-implement CLI for sentential logic

-implement GUI for sentential logic

-implement first order logic

-implement second order logic

-implement CLI for predicate logic

-implement GUI for predicate logic

-implement basic theorem proving for sentential logic

-finish theorem proving for sentential logic

-implement basic theorem proving for predicate logic

-finish theorem proving for predicate logic

-clean up CLI and GUI for final release

-finalize program structure for final release

Prioritized content

Note: Priorities are interleaved between both features and architectural concerns so that we have one single backlog to pick items from when starting development work.

First Pass – 2019-JUL-24

1. Create documentation strategy
    a. Feature/story content DoD: feature dev'd, unit tests pass, demo-able, and impacted documents updated.
2. Define sprint length and figure out how to track in GitHub (2 weeks, use milestoens?) Make release candidates how often: every 4 sprints?
3. Update architectural documentation
4. Define initial branching strategy: DEV as master. <features> for development and testing, demos for end-of-sprint demos if needed, REL<release #> for release candidates.
5. Research better program structure for release candidates
6. Define and document initial CLI approach
7. Define and document initial GUI approach
8. Define unit testing framework: open source or in-house?
9. Make DEV branch's functionality match the prototype's, at a minimum, so we have a good starting point.
10. binary operator distribution
11. double negation
12. order of operations when parens aren't specified (nice-to-have?)
13. implement CLI for sentential logic
14. implement basic theorem proving for sentential logic
15. finish theorem proving for sentential logic
16. Choose GUI toolkit, at least for initial development.
17. implement GUI for sentential logic
18. implement first order logic
19. implement second order logic
20. implement CLI for predicate logic
21. implement basic theorem proving for predicate logic
22. finish theorem proving for predicate logic
23. implement GUI for predicate logic
24. clean up CLI and GUI for final release
25. Determine if we want to allow multiple instances of ManipuLogic to run simultaneously
26. finalize program structure for final release

Project Strategy

From the goals and backlog item prioritization it is clear that the strategy roughly follows this timeline:

1. Create initial documentation and development strategy by defining the organization and granularity of project documentation and getting basic architectural things out of the way that have an impact of WoW considerations so we can get going on dev ASAP.
2. Get basic SW architectural considerations out of the way in order to provide a development framework so that development will steadily meet the project goals. Of course this will evolve over time with discoveries made during development.
3. Since the prototype worked well for basic symbol manipulation, this is a good place to start. So within our new WoW and architectural framework in the DEV branch, and with our unit testing in place, we can make a cleaner and more extensible version of the prototypes feature set in the DEV branch, further improving upon the "backbone" of code that the loftier project goals will be built on.
4. Start working on features by getting code and unit testing for the "guts" first, then develop CLIs for E2E testing, then GUIs as appropriate for user testing and demos.
5. **NOTE:** This will evolve as the non-functional-requirements' considerations start getting violated during development, e.g., if the program becomes noticeably cumbersome to use or operates slowly. Architectural concerns of this nature will of course be considered during all of development, but refactoring and some re-design will be inevitable. As features roll in, these things will change as well, e.g., maybe we'll tease out the reading in of saved "logic files" to a script that ManipuLogic runs only when needed, due to software role/responsibility considerations.

Documentation Strategy

Project Docs:

1. Project Plan – To be updated as-needed when something in the scope of the overall project changes: e.g., major priority shifts, adding a new type of document, etc…
2. ManipuLogic System Design – To be updated when major architectural designs are made for the ManipuLogic program and its environment, environmental interactions, or when any peripherally-related software gets updated.
3. Sub-system/Component Design Documents – Created at the start of a new major sub-system in ManipuLogic, or in any related software that's developed in-house. Updated when a sub-system or its interactions are updated in the codebase.
4. Test Executions Results Document – To be updated as unit test results roll in from checkins to the _DEV and _REL<release #> branches.

Way of Working (WoW)

1. Definition of Ready (DoR) for feature stories: At least one user-level acceptance criterion (AC) defined and sized (small, medium, large, huge) and of not of huge size. Potential documentation impacts noted. Potential subsystem noted so they can be tagged for unit testing.

2. DoR for architecture stories: Scope of investigation and stopping criteria defined. Sizing requirement same as DoR for feature stories.
3. DoD for features stories: all acceptance tests pass and are documented if appropriate, all relevant unit tests pass, unit tests tagged for smoke testing pass, all impacted documents updated, issue moved to done in GitHub Kanban.
4. DoD for architecture stories: all acceptance tests pass if applicated and are documented if appropriate, all impacted documents updated, new stories from arch decisions created, issue moved to done in GitHub Kanban.
5. Sprint length will initially be two weeks.
6. Branching strategy will be as follows: _DEV will be the master; work directly out of it only for small patches. For features/stories create either a feature or sprint branch with the name _<FEATURE> or <SPRINT_TAG>, to die off when the feature or sprint is done and all code changes from the branch is merged into _DEV. _REL<release #> will be cut for demos/release to the public as appropriate (~every 4 sprints, but subject to change if appropriate). <release #> will start at 1001 and simply increment by 1 with each subsequent release candidate. _REL branches will never die for posterity's sake.
7. Testing strategy: Unit tests will be run daily on feature branches (locally, probably). A unit test suite will be tagged as "smoke" or "not smoke". All "smoke" test suites in _DEV are to be run upon any changes to _DEV, whether via direct checkin or a merge from a _FEATURE or _SPRINT branch. Acceptance testing will be done at a user level, E2E, as each story is to be completed in _DEV.