

Automation of Multi-stage Data Processing in Neuroimaging

Ben Jarvis^{1,2}, Hu Yang¹, Josh Stern¹

University of Minnesota, ¹Dept. of Neurology, ²Dept. of Computer Science



Background

The Registration Testbed (RTB) was developed within an ongoing project evaluating multiple software packages for inter-subject image registration.

The project's experimental goal is a comprehensive assessment of the performance of particular versions of these packages (currently 8) for particular choices of goodness-of-warp metrics (GoWs >10), images (>50), and neuroanatomical regions of interest.

Software supporting the project must integrate existing, stand-alone programs for warping and image pre-processing with new software for:

1. Reading the volume deformation warps created by these program in their native format
2. Computing various GoW metrics involving these warps
3. Performing data analysis on the multivariate indexed results.

Each GoW computation may itself depend on ancillary data with elaborate pre-processing requirements, e.g., automated tissue segmentations, and functional data analyses.

Requirements and Desiderata

Automation and Computational Abstraction

We want to automate processing tasks and describe them at a high level of abstraction - e.g. "Apply a set of GoWs to all possible warps generated by applying a set of Warping algorithms to all ordered pairs of source and target image volumes and summarize the average performance, tabulated by algorithm". Conversions between known data formats should be handled automatically in the process.

Re-use of intermediate data

We want to efficiently re-use intermediate results just in case they are still valid - e.g. if the definition of one GoW changes then the software should understand that recomputing the previous example only involves recomputing results involving that GoW and those results can also re-use the existing warps.

Data cataloguing

Allow search and retrieval of data objects and results based on patterns of semantic features.

Audit support

Associate each data object with the processing chain that created it, the logs that were recorded, its filename (where relevant), creation date, and a unique cryptographic hash to detect identity and differences in the presence of filename changes.

Developer Support

Maximize convenience of developing new processing components and integrating new and old components into the framework. In particular, support plugin components in addition to stand alone executables, and allow non-programmers to easily incorporate additional stand alone executable components. Minimize the complexity of specifying correct processing pipelines and help prevent specification of incorrect ones.

Software Design Features

Metadata Types

Data objects are associated with metadata types. Each type has a unique name and a map from property names to data values. Derived types are supported within single inheritance hierarchies. Tuples, arrays, and sets of data values are also supported as data values. Common properties of data objects include the filename of the objects location on the file-system (or blob location in a database), a cryptographic hash of its contents, the date it was entered into the repository, and the method used to create it. Type hierarchies are read from editable configuration files at runtime (Fig. 1).

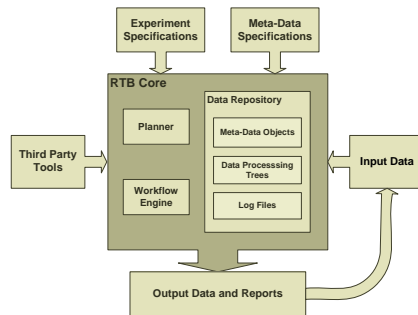


Fig 1: Testbed Application = Core + User Specs + Data

Object repository

Metadata for objects is stored in a repository implemented using a relational database, so sets of objects (including analytical results) can be retrieved based on featural patterns of their metadata (Fig. 2).

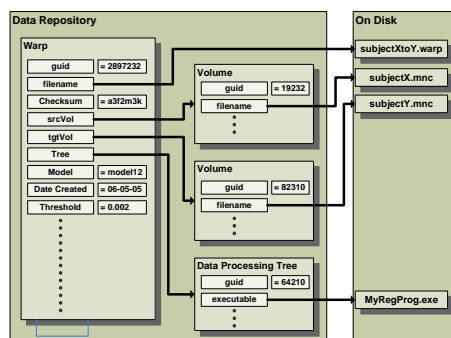


Fig. 2: Example of Metadata Objects in Repository

Constructor Rules

Runtime readable configuration files describe the valid rules for making new data objects and results from existing ones. A data constructor combines:

1. A logical specification describing constraints on the relationship between the metadata properties of a set of input objects and an output object
2. A recipe for executing a software routine that creates the real data object file (or stores the intended analytical result in a database).
3. Various types of recipes including:
 - a) Running a stand-alone executable or script with given command-line arguments
 - b) Loading and running a C/C++ plugin modules which takes RTB key/value mappings directly as arguments.

Data Processing Trees

A complete recipe for building a new object of a given type and property map from existing objects stored in the repository has the form of a tree. The leaves of the tree correspond to objects already existing in the repository and the internal nodes of the tree correspond to constructors with inputs represented by sub-branches. A valid tree is one that is:

1. Type correct in terms of the inputs and outputs at every node
2. Satisfies a set of equational constraints - given by the constructor specifications - over the key/value property maps for the specific objects in the roles of input and output arguments for that constructor at each given tree position.

Computational cost estimates are associated with every tree based on the cost estimates for the executable routines correspond to the constructors appearing in the tree.

Computational Planning and Execution

RTB will always search for a complete and valid processing tree prior to executing the computations described by the tree's internal nodes. This separation of the planning an execution phases of computation has the following benefits:

Computational cost savings based on searching for the least cost valid processing tree. This implies re-using existing data objects wherever possible, and avoids speculative execution that will not ultimately satisfy the new object request. Storage of the processing tree itself as an object that can be independently inspected and used in pipeline, development, debugging, and auditing.

The algorithm for generating valid processing trees based on metadata type/feature requests involves rule-based backward chaining and unification. It is reminiscent of execution in the Prolog language but differs in two fundamental respects:

RTB uses an iterative deepening A* search algorithm to find cost optimal trees whereas Prolog uses depth first search. Prolog is a relational programming language that generates work as side effects that are intertwined with the searching computation itself, while the planning module of RTB is a purely declarative meta-language that searches for procedural recipes, relying on the underlying object code of the recipes themselves to do the desired work.

Comparison with Other Approaches

Scripting

Using languages such as Perl, Python, Ruby, Tcl, etc.

Advantages:

- Familiarity
- Flexibility

Disadvantages:

- Need to hand code and self-document everything.
- Poor reusability of components
- Error prone.

Workflow and "Pipeline" Software Packages

Many packages are available - websites are given in the reference section for Fiswidgets, LONI, Kepler, Pegasus, Pegasys, SciRun, and Taverna.

Advantages:

- Greater maturity of code and features as software packages (e.g., existing GUI designs, integrate many existing third party tools, multiprocessor scheduling, networking support, etc.)
- Often Java based and more oriented towards the needs of non-programmers. Fiswidgets and LONI come with support for a variety of popular Neuroimaging tools.

Disadvantages:

- Limited support for reuse and/or invalidation of previously constructed objects
- Computational modules must be implemented by command line programs
- No support for complex, heterogeneous processing trees
- Lack of validity constraints (i.e. simplistic or missing notions of types)
- No Automatic construction of processing tree designs based on specifications
- Database integration and auditing is usually more ad hoc

References

- Fiswidgets <http://grommit.Irdc.pitt.edu/fiswidgets/>
- Kepler <http://kepler-project.org/>
- LONI <http://www.loni.ucla.edu/wiki/bin/view/Pipeline/>
- Pegasus <http://pegasus.isi.edu/>
- Pegasys <http://bioinformatics.ubc.ca/pegasys/>
- SciRun <http://software.sci.utah.edu/scirun.html>
- Taverna <http://taverna.sourceforge.net/>

RTB Testbed supported by NIH Grant P20 EB002013