SASSY Concepts

July 2026

Contents

1 Introduction **5**
- 1.1 Scope . . . 5
- 1.2 Overview . . . 5
- 1.3 Audience . . . 5

2 Process View **7**
- 2.1 Preliminary Analysis . . . 7
  - 2.1.1 Mind Mapper . . . 7
  - 2.1.2 Glossary . . . 7
  - 2.1.3 Directory . . . 7
  - 2.1.4 Library . . . 7
  - 2.1.5 Bibliography . . . 7
  - 2.1.6 Spreadsheet . . . 7
  - 2.1.7 Diagramming . . . 8
  - 2.1.8 Mapping Tool . . . 8
- 2.2 Modelling . . . 8
- 2.3 Requirements . . . 8
  - 2.3.1 Functional Requirements . . . 8
  - 2.3.2 Environmental Requirements . . . 8
  - 2.3.3 Quality Requirements . . . 8
- 2.4 Requirements Analysis . . . 9
- 2.5 Scheduling . . . 9
  - 2.5.1 Waterfall . . . 9
  - 2.5.2 Incremental . . . 10
  - 2.5.3 Spiral . . . 10
  - 2.5.4 Agile . . . 10
- 2.6 Tactical Analysis . . . 10
  - 2.6.1 Architectural Design Patterns . . . 10
  - 2.6.2 Tactics . . . 10
  - 2.6.3 Architectural Decision Records . . . 11
  - 2.6.4 Automated Design . . . 11
  - 2.6.5 Responsibilities . . . 11
- 2.7 Architectural Design . . . 11
  - 2.7.1 Conceptual Model . . . 12
  - 2.7.2 Logical Model . . . 12
  - 2.7.3 Physical Model . . . 12
  - 2.7.4 Modules . . . 12
  - 2.7.5 Interfaces . . . 12
- 2.8 Actions . . . 13
  - 2.8.1 Tactic Selection . . . 13
  - 2.8.2 Tactic Optimisation . . . 13
- 2.9 Reports . . . 13
  - 2.9.1 Omissions . . . 13
- 2.10 Views . . . 13

2.10.1 Management View . . . 13
2.10.2 Application View . . . 13
2.10.3 Database View . . . 14
2.10.4 Network View . . . 14
2.10.5 Equipment View . . . 14

3 Structural View 16
3.1 Infrastructure Layer . . . 16
3.1.1 Common Facilities . . . 16
3.1.2 Development and Testing Support . . . 16
3.1.3 Sable . . . 17
3.1.4 ICE . . . 17
3.1.5 Spelling Checker . . . 17
3.1.6 Qt GUI Library . . . 17
3.1.7 GUI Testing . . . 17
3.1.8 RDF for C++ . . . 18
3.1.9 RDF Utilities . . . 18
3.1.10 RDF GUI . . . 19
3.1.11 Reasoner . . . 20
3.2 Utilities Layer . . . 21
3.2.1 Schema Tool . . . 21
3.2.2 Gayal . . . 21
3.2.3 Rule Engine . . . 21
3.2.4 Lexicon . . . 22
3.2.5 Glossary . . . 23
3.2.6 Transformers . . . 23
3.3 Input and Output . . . 24
3.3.1 Layout Planner . . . 24
3.3.2 Entry Form Language . . . 24
3.3.3 Static Form Generator . . . 25
3.3.4 Dynamic Form Generator . . . 25
3.3.5 Mind Mapper . . . 25
3.3.6 Document Generator . . . 25
3.3.7 Diagram Generator . . . 26
3.3.8 Table Planner . . . 26
3.3.9 View Planner . . . 27
3.3.10 English Text Planner . . . 27
3.3.11 Natural Language Generator . . . 28
3.4 Processing . . . 29
3.5 Control . . . 29

4 Ontologies 31
4.1 Schemas . . . 31
4.1.1 Analysis Schema . . . 31
4.1.2 Architectural Decision Record Schema . . . 31
4.1.3 Bibliography Schema . . . 31

4.1.4 Core Schema . . . 31
4.1.5 COTS Schema . . . 31
4.1.6 Diagram Description Schema . . . 31
4.1.7 Directory Schema . . . 31
4.1.8 Document Schema . . . 32
4.1.9 Entry Form Schema . . . 32
4.1.10 GIS Schema . . . 32
4.1.11 Glossary Schema . . . 32
4.1.12 Library Schema . . . 32
4.1.13 Modelling Schema . . . 32
4.1.14 Requirements Schema . . . 32
4.1.15 Rule Schema . . . 32
4.1.16 Scheduling Schema . . . 32
4.1.17 System Design Schema . . . 33
4.1.18 Tactics Schema . . . 33
4.1.19 View Schema . . . 33
4.2 Reference Ontologies . . . 34
4.2.1 Lexicon . . . 34
4.2.2 Quality Attributes . . . 34
4.2.3 Standard Tactics . . . 34
4.2.4 Preferred COTS . . . 34
4.2.5 Data Entry Form Styles . . . 34
4.2.6 Diagram Styles . . . 34
4.3 Project Ontologies . . . 35
4.3.1 Catalogue . . . 35
4.3.2 Analysis . . . 35
4.3.3 Glossary . . . 35
4.3.4 Directory . . . 35
4.3.5 Library Catalogue . . . 35
4.3.6 Bibliography . . . 35
4.3.7 GIS . . . 35
4.3.8 Existing System Model . . . 35
4.3.9 Requirements . . . 36
4.3.10 Schedule . . . 36
4.3.11 Tactics . . . 36
4.3.12 COTS . . . 36
4.3.13 ADR . . . 36
4.3.14 System Design . . . 36
4.4 Temporary Models . . . 37

1 Introduction

This document is intended to give an early overview of the design of **SASSY**. Eventually it will be replaced by a report from **SASSY** itself.

One might call it a "concept for a design".

1.1 Scope

The components that might become part of **SASSY** are enumerated and briefly described. The description is in terms of the software architecture process. At this early stage the descriptions are more an indication of the intentions rather than any sort of specification.

It should be noted that the high level design process can be applied to other domains besides software.

1.2 Overview

The first part describes the system from the perspective of its users. The emphasis is on those components that the user directly interacts with.

Part two goes into more detail of the underlying support modules. It works from the bottom layer of the system and proceeds up to the top layers that were covered in part one.

The third part describes the data that supports the processing.

1.3 Audience

Anyone wanting an early view of what **SASSY** entails.

Part I

2 Process View

In this section the components that a user of **SASSY** would interact with are outlined. This is not meant to be a set of detailed use cases, but just a broad brush overview.

2.1 Preliminary Analysis

In this phase of the project the task is to learn as much a possible about the domains that the proposed system will be working in. This is not really part of the architecture development process, but the sooner we can capture the information the quicker we can get the real work underway.

SASSY will provide a set of tools to help organise the information as its gathered:

2.1.1 Mind Mapper

Use this to map out ideas, concepts and their relationships. This tool will be central to the preliminary analysis process. An array of other tools will be available to help manage and organise the data:

2.1.2 Glossary

Having good definitions of the domain specific terminology is a vitally important part of P.A. The glossary enables these terms to be captured into a form that can be used throughout the entire project.

2.1.3 Directory

Knowing who is responsible for various aspects of the project will save an enormous amount of time. A listing of the people and organisations involved, and their relationships is the job of the directory.

2.1.4 Library

The P.A. process will most likely capture a large amount of documents, diagrams, and images. Video and audio recordings might also be involved. These need to be stored in a repository with a good catalogue of what it contains.

2.1.5 Bibliography

It is useful to keep track of the books and papers, etc. that are used to guide the design. A bibliography helps others locate them if they need to find out more on the subject.

2.1.6 Spreadsheet

There will often be tabular data that needs to be collected and analysed.

2.1.7 Diagramming

A tool for creating diagrams during the information gathering process is likely to be useful. **SASSY** will have some tools for automatically creating diagrams, or some existing tool can be used.

2.1.8 Mapping Tool

It is often quite useful to know where everything is located. Capturing and maintaining spatial information will require some form of mapping tool.

2.2 Modelling

For projects that are replacing, or interfacing to, some existing systems it is important to capture the essential details of those systems. It really annoys users when they discover the shiny new replacement is unable to do what the old system could.

Add lots more to this

2.3 Requirements

For large systems it might be useful to subdivide the requirements by domain. **SASSY** will provide a data entry program to enter the domain and the other aspects of each requirement.

Each requirement should come with indicators for their importance and their urgency. Links to the preliminary analysis would help with understanding of the requirements.

2.3.1 Functional Requirements

These are the requirements that specify what the system must be able to do for its users. For large systems, and especially in domains that are still being defined, the functional requirements can be quite fluid. It is to be expected that some requirements will disappear and new ones will get added as the system is delivered.

2.3.2 Environmental Requirements

These are typically constraints on the design. They might, for example, insist that only open source products can be used. They might also specify that the product is to be used in aerospace or remote sites.

2.3.3 Quality Requirements

The quality requirements typically have the most impact on the architectural design of a system. Functional requirements can usually be accommodated by changing or adding application programs and the database tables, whereas

changes to the quality requirements can often require significant modifications to the system.

The **SASSY** system will provide a catalogue of common quality attributes that can be used as a check list for these requirements.

The quality requirements can include things that influence the design of the system, such as a need for resilience, and things that influence the development process itself, such as *Time to Market*.

Since the quality attributes are a well defined set, we can express these requirements using a formal language. This can then be transformed into a symbolic form (RDF) for automated processing.

2.4 Requirements Analysis

This task checks the requirements looking for those that are impossible, infeasible, or contradictory. Also watch out for those that are vague or ambiguous.

The quality requirements can be automatically checked for potential problems since we should have them in symbolic form.

2.5 Scheduling

The style of scheduling can affect the architectural decisions. When a waterfall style is appropriate it is probable that all the requirements are well defined and the architecture can be well defined. However, at the other end of the scale, agile is, by definition, used when the full set of requirements cannot be determined up front, a more flexible architecture might be a better approach.

Ideally it should be possible to export the project details into a project management tool. However, software development has some unique characteristics that make all but the most expensive tools less than adequate for planning a software project.

By using probabilistic estimates for duration and the chances of bug fixes and rework, estimates using Monte Carlo simulations can give a much better guide to progress.

2.5.1 Waterfall

The waterfall development model does each of the development phases in order, completing one before moving on to the next. Analysis, then design, then coding, and finally testing.

It works OK for small projects where everything is well understood. Leaving the testing to the end can be problematic. One common improvement is to have the unit testing done with the coding. The main testing is then typically just a confirmation it all works.

2.5.2 Incremental

The incremental development model develops software in small, manageable parts called increments, allowing for early delivery of functional software and the ability to adapt to changing requirements. This approach promotes frequent testing and feedback, making it easier to address issues and improve the final product.

2.5.3 Spiral

The spiral development model addresses the problem that the delivery of urgent functionality is delayed because the effort is being spent on the supporting infrastructure.

By starting with the urgent functionality and just enough infrastructure to get it working, users and management are much more likely to continue support. That functionality may need to be reworked with the new infrastructure when it gets delivered later.

2.5.4 Agile

The agile model is most suitable for scenarios where the final design cannot be determined, but software support for the business is urgently required.

A common solution in this case is an off-the-shelf framework that has a ready-made architecture. This is a viable alternative if you can be confident you won't have to go beyond what the framework can provide.

In order to not have to revisit the architecture, no matter what the functional requirements are, the design tends to things like micro-services or event driven design. These are extremely flexible architectures that can be adapted to most scenarios. The cost is significantly more complex management of the running system.

2.6 Tactical Analysis

2.6.1 Architectural Design Patterns

SASSY will have a library of well known architectural design patterns. Each pattern will have links to the quality attributes with an indication as to how well they are supported.

2.6.2 Tactics

Tactics are the building blocks of the architectural decision process. An architectural pattern might involve several tactics.

A tactic can be considered as an option for resolving one or more requirements.

SASSY will have a library of well known tactics, probably as a part of the architectural design pattern library.

COTS Using COTS (Common Off The Shelf) software, either commercial or open source, is a desirable tactic as it avoids a lot of development time. Of course there is a down side - interfacing to the product can put additional complications into other modules.

2.6.3 Architectural Decision Records

Trying to satisfy many requirements simultaneously will result in a lot of tactics to decide between. Tactics can be independent, reinforcing, or incompatible, to varying degrees.

The primary task of the architect is to decide which set of tactics will provide the best solution for the requirements. It is important that these decisions and the reasoning that went into them is recorded since the decisions may need to be revisited if the requirements evolve over time.

The ADRs will become part of the knowledge database. A data entry program will enable them to be entered, reviewed and modified.

2.6.4 Automated Design

If the tactics can be codified, and the requirements expressed as constraints it might be possible to automate some of the process.

2.6.5 Responsibilities

Each tactic implies a set of responsibilities that must be satisfied by the resultant system, or the development process that creates it.

Allocating these responsibilities to components and development tasks is the nub of the architectural design process.

2.7 Architectural Design

The architecture task is further divided:

- A conceptual architecture with a focus on identification of components and allocation of responsibilities to components;
- A logical architecture with a focus on design of component interactions, connection mechanisms and protocols, interface design and specification, and providing contextual information for component users;
- An execution architecture with a focus on assignment of the runtime component instances to processes, threads and address spaces, how they communicate and coordinate, and how physical resources are allocated to them.

2.7.1 Conceptual Model

The process involves subdividing the set of responsibilities into smaller and smaller modules. This stops when each module can be completed in a reasonable time by a single team.

Responsibilities that are the province of the development process are allocated to tasks.

A text based system design language is likely to be the means of performing this task.

2.7.2 Logical Model

The process involves adding descriptions for the interfaces between modules. The descriptions should include the data set and protocol that is communicated over the interface.

In order to be able to confirm that a module has all the inputs it needs to satisfy its outputs a description of data transformations will be required.

2.7.3 Physical Model

The modules are now allocated to libraries and programs, and those are allocated to hardware.

2.7.4 Modules

The main danger when developing modules is deep coupling, where a module depends on the internal workings of another module rather than its published interface.

Using knowledge of the internal workings of other modules might seem to be beneficial, but if there is any delays in communicating this information either via the operation of the programs or just chatter between the developers things can become a chaotic mess.

Layers A common design tactic is collecting the modules into layers. The bottom of the pile contains the low level infrastructure, up to the top where the user applications reside.

There is a need to be careful not to spend all your time budget on the lower layers and not creating any value for the users. It might be worth having a range of capabilities for the low level modules so that they can support some functionality early in the project.

2.7.5 Interfaces

With the definition of the interfaces it becomes possible to start building test harnesses for each module. These test harnesses enable each module to be built

without relying on other modules, though it does work best within a single layer.

2.8 Actions

Various actions become automatable once the design data is in machine processable form.

2.8.1 Tactic Selection

The tactics database will include with each tactic information about how it relates to quality attributes. By using this and the attributes of the requirements it should be possible to automatically generate a preliminary list of candidate tactics.

2.8.2 Tactic Optimisation

It might be possible to use a constraint solver, or something similar, to find the optimum set of tactics.

2.9 Reports

I imagine there will be a lot of reports that will come to light as development proceeds.

2.9.1 Omissions

One of the benefits of having everything in a knowledge database is that we can design reports that highlight anything that is missing. A requirement that is not addressed by a tactic, responsibilities not allocated to a task or a module, and similar omissions.

2.10 Views

Having created the design it is now time to report what has been done. A set of views will be available for different categories of users. For example, a network engineer might want a network view showing the equipment that is to be interconnected and the expected traffic volumes.

2.10.1 Management View

The manager's role is to monitor and control quality, progress, change and knowledge. The view will include data about testing, schedule and tasks, changing requirements and access to the knowledge database.

2.10.2 Application View

The application view is the primary input for the developers. It will provide a complete report of a module, its interfaces and required data transformations.

2.10.3 Database View

The database view would include a full description of the data, including types row counts and transaction rates.

2.10.4 Network View

The network view also needs to include the data types, as well as volume and rate, but this time it is for interfaces that involve separate executables on separate hardware.

2.10.5 Equipment View

This view is for determining the hardware that the programs need to run on. It might be little more than the load it will put on existing machines, through to a catalogue of dedicated equipment.

Part II

3 Structural View

In this section we describe the components from the bottom up.

3.1 Infrastructure Layer

3.1.1 Common Facilities

This is a library that contains a collection of small classes that are generally useful in most large projects.

sx An exception class that uses the stream operator («) to construct the error message. It also has severity as a parameter.
stringy A set of useful string functions and a class for file paths.
fdstream A file stream class that uses the file descriptor number as its parameter.
ipc Semaphores for coordinating programs.
options A C++ wrapper for the GNU getopt(3) function.
singleton An alternative way of doing singleton classes that provides more control over their construction.
plugin Classes for managing plug-in libraries. The plugin library can be either *Autorun* where it can be removed after initilisation, *OnDemand* which can be removed if no longer needed, or *StayResident* which remain for the duration of the program.
proc Classes for managing child processes.
udpsocket A C++ wrapper for a UDP socket.
log Creates formatted or unformatted messages which can be passed to a variety of sinks including the output and error streams, files, a logging server, or the system log.
xml A C++ wrapper for the libxml2 C library.
xini Classes for an XML based configuration file.

Status: A well tested component used in several projects.

3.1.2 Development and Testing Support

The cdi library is not normally part of a delivered system. It contains facilities for testing and execution tracing.

discover Classes and macros that enable a test harness to locate an object deep inside the internal structure of a program. This can be used for testing objects in situ.
testmgr Classes for testing. It uses the instantiation of the test objects to set up the test cases. Supports asynchronous testing.
trace Classes for tracing the execution of programs.

Log Server A simple server for receiving UDP log messages.

trc2dot A program for converting function call trace logs into collaboration diagrams. Can show an animation of the call sequence.

Status: The tracing could use some work.

3.1.3 Sable

SableCC is a Java program that takes a grammar and creates a parser for it. It has two parts, the second of which generates the appropriate code for the parser language. The second part, for non-Java parsers, is in an additional, and perhaps unsupported package. I have had to make a modification to it to fix its C++ generation for more recent versions of the language that is a bit more strict wrt constants.

A possible enhancement is to use Sable to create a C++ parser for a grammar that describes Sable's grammar. This could then be extended to create a C++ replacement.

Status: Stable product. Possibility of conversion to C++.

3.1.4 ICE

The ICE package from *ZeroC* is a lightweight CORBA implementation. It is good for building servers, especially across languages.

Status: Stable product.

3.1.5 Spelling Checker

Using **SASSY** involves a lot of text entry. It is probably a good idea to correctly spell things to avoid errors and duplications.

Status: Seems like a good idea.

3.1.6 Qt GUI Library

The GUI programs use the Qt library. The library is used directly without the supporting design and build utilities.

Status: Stable product.

3.1.7 GUI Testing

The **VICI** project included a module for testing GUI programs. It adds a wrapper to each GUI component so that it can be manipulated by the test harness. This avoids the problem of trying to address widgets based on their screen location.

Various means are available for creating the tests, including Lua scripting for the most complex tests.

The module needs to be copied to **SASSY** and completed, at least for the widgets being used. (Need to confirm that it will integrate with QML.)

Status: Have a prototype.

3.1.8 RDF for C++

This is a wrapper for the *Redland* C library for RDF. It has classes for resource, literal and blank nodes; statements (triples; models; parsers; and serialisers. It supports RDF/XML and *PostgreSQL* storage but others that are supported by *Redland* could be added easily. It also supports temporary models that only exist in memory.

A recent addition is a subtype of the resource node for anonymous nodes. They use a base 64 encoded UUID as their identifier.

Status: Working version.

3.1.9 RDF Utilities

Catalogue An RDF/XML file is used to store catalogues of RDF models. This contains information on how the model is stored, any submodels that a model includes (see below), and the definition of the prefixes used in the model. The RDF Utilities library provides a means of opening models via their name.

Status: Working version.

Compound Models A model can include submodels. The submodels are loaded into a read-only in-memory model. The compound model then applies actions to the two models.

Status: Working version.

Containers Wrapper classes are included that map RDF lists on to C++ std::list and RDF sequences and bags on to std::vector.

Status: Needs enhancements.

Boundaries Each model can have a set of nodes defined that form a boundary for certain operations. For example, deletion of a region needs to be constrained to avoid deleting things you meant to keep.

Status: Planning.

Views When accessing large models it is easily possible to exceed the memory limits for programs if the entire model is loaded. A view can be defined that enables small regions of a model to be loaded.

Status: Planning.

3.1.10 RDF GUI

The *rdfgui* program provides a means for inspecting and editing RDF models. It has the following tabs:

Catalogue This lists the models in the current catalogue (selected via the XINI configuration file). It has an interface for adding and removing models, creating new models, and setting the submodels for a model.

Its main use is selecting the model to open. Multiple models can be open concurrently.

Models This tab allows the user to select which open model is current in the other tabs. It has a set of buttons for perform operations on the current model: save, reload, clear, close, export, import, and insert. (Import copies the data from a file or database, while insert copies it from another model).

Prefixes This tab allows the user to set and edit the prefixes associated with a model. Prefixes are a short hand for the URIs that define resource nodes. They are generally not expected to be defined outside the model, and could be different for the same URI indifferent models, though this is not recommended for **SASSY**.

Statements This is the main tab for viewing and editing RDF data. It has three columns, for subject, predicate, and object of each statement.

Visual This tab displays a small view around the currently selected node as a graph. It is possible to navigate over the model from this interface, but editing must be done from the *statements* tab.

Schema This tab allows the user to define a schema. A class hierarchy and a property hierarchy can be set up and edited.

Add details for each panel of the schema tab

Query & Rules The last tab has a panel for entering and running SPARQL queries on the current model. It has panels for selecting and running rule engines. It has a panel for editing and parsing rules for rule engines. (See the section on *Rule Engines*)

Form Editor Available from the tool bar is a forms based interface for entering data into a model. The forms are dynamically constructed from the schema.

Status: Working version.

3.1.11 Reasoner

The OWL language, which is designed for representing Semantic Web ontologies, allows the user to introduce data types into ontologies (in addition to the usual mechanisms for describing "abstract" classes, such as boolean conjunctive, quantifiers and cardinality constraints). In order to use these ontologies one should have an efficient, robust reasoner, which supports all the features of the language.

In order to use a reasoner designed for OWL with RDF the properties need to be segregated into data properties and object properties. The schema in *rdfgui* provides that distinction.

Status: Needs more testing.

FaCT++ FaCT++ can be used for the following tasks:

- Checking the consistency of an ontology;
- Checking the satisfiability of a single concept / group of concepts;
- Checking the subsumption relationship between two concepts;

FaCT++ provides full reasoning support for the *SHIF(D)* Description Logic. More precisely, it supports:

- Intersection, Union and Complementation of concepts;
- Universal and Existential restriction on roles;
- Transitive, Functional and Inverse roles;
- Integer and String data types.

Status: Stable product.

3.2 Utilities Layer

3.2.1 Schema Tool

The schema tool from *rdfgui* will be extracted into a library that can be added to other GUI programs so that the schemas can be examined and modified.

Status: Working version in *rdfgui*.

3.2.1.1 Catalogue Tool The catalogue tool from *rdfgui* will be extracted into a library so that all GUI programs can have this functionality.

Status: Working version in *rdfgui* which needs some improvements to its user interface.

3.2.2 Gayal

Gayal is a bit like a compiler for RDF. The *gayal* program is used at compile time to generate code that parses RDF based on an RDF schema and generates C++ structures that parallel the RDF data.

The *gayal* program uses plug-in libraries to control what is generated. One plug-in generates a document that describes the schema; another produces C++ structures for use with a processor the implements the **visitor** pattern.

A library, *libgayal*, provides an interface for the resulting program to access Gayal's functions.

Status: Working version.

3.2.3 Rule Engine

The **SASSY** rule engine rule uses a SPARQL query to obtain a result set, each member of which is a set of values from the RDF model. The result set can be sorted and filters applied to remove some members. If a member contains a node that is the basis for a list or sequence an iterator can be used to cycle through its entries. Finally the values can be used in operations which can add or remove statements, apply user supplied functions or call other rules.

The rules are stored as RDF. It is therefore possible for the rule engine to modify its own rules (but not encouraged).

The rule engine module includes the following components:

libreng This library loads the rules from an RDF model and creates an interpreter which can then be run to update the model. This library uses Gayal generated code to parse the RDF representation of the rules.

rule-app This is a simple wrapper around *libreng* so that it can be used by *rdfgui*.

rule-text This program also uses Gayal generated code, in this case to create a text version of the rules. This makes it much easier to view and edit the rules.

rule-parser This program parses the rule text (see previous) to create an RDF representation of the rules. It uses code generated by *SableCC* to perform the parsing.

Status: Working version. Needs additional built in functions.

3.2.4 Lexicon

The Natural Language Generator (NLG) requires a lexicon in order to determine the part of speech and other attributes of words. The one provided with the original code was sufficient for testing, but not for real world applications. Hence **SASSY** includes a lexicon.

The lexicon (RDF in *PostgreSQL*) is populated during installation from three data sources:

Wiktionary The primary source for this dictionary is the English language open dictionary. This data has been processed into JSON by *Wiktextract*. It is necessary to filter out the non-English words and proper names.

Wordnet This data comes as RDF in Turtle form, which we can read. It is words linked by various relationships, such as synonyms.

Roget The thesaurus data is parsed from a slightly edited *Gutenberg* version of *Roget's Thesaurus* which is rather dated, but will do for now.

The lexicon module has several components in addition to those required to install the data. A GUI for viewing the data for a word and an *ICE* based client and server:

lexgui Shows the part of speech and other attributes of a selected word. The various meanings (glosses) are listed along with links to synonyms, etc and the thesaurus entry.

lexsvr Accessing the relational database every time a word is used is too slow, so on first use the word and the data required by the NLG are loaded into a memory based table. This data is then made available via an *ICE* interface to a client program.

Status: Needs a rebuild.

3.2.5 Glossary

The English Text Planner needs to be able to map RDF URIs to English words. The *glossgui* is used to create an RDF model that maps URIs to words along with their part of speech and semantic strength.

Status: Initial working version.

3.2.6 Transformers

The data used by **SASSY** needs to be in two different forms. A symbolic form (RDF) for processing, and a natural language form for display and reports. Holding both forms in its database is likely to create issues for maintenance and accuracy. A means of transforming from one form to the other is preferred.

A natural language form is too difficult to process, or parse, so **SASSY** will use formal languages (i.e. with a grammar) instead.

The job of a transformer is to perform the transformations between the two representations. It will use *Gayal* to parse the RDF data and *Sable* to parse the text. In each case code will then create the corresponding alternate form.

It might be possible to annotate the grammar with RDF symbols so that a code generator can automatically create the code to generate the alternate forms.

Status: Seems like a good idea.

3.3 Input and Output

Key functionality of **SASSY** is the ability of getting data into and out of its knowledge database in user friendly ways. Hand editing RDF data does not count!

Input will mostly be done through data entry forms or the text of a formal language. The data entry forms will be generated automatically based on the RDF schemas.

Output will be available for a form interface, or formal language text, or (hopefully) via English language reports.

3.3.1 Layout Planner

Data entry forms and diagrams should be generated automatically. There will be a lot of them and manually editing them will inevitably lead to them being left behind as the system's design evolves.

The layout planning will be done in three stages:

Focus The idea is to provide a strong backbone to the layout. The RDF data that was selected for the layout will be searched to find the shortest path with the largest semantic strength.

Rhetorical Structure The Rhetorical Structure Theory algorithm will be used to sort the nodes of the RDF graph.

ELK The ELK (Eclipse Layout Kernel) will be used to arrange the nodes into a two dimensional layout.

ELK is a Java library that will be interfaced to the C++ code using *ICE*. It will therefore run in a server process.

Status: Planning.

3.3.2 Entry Form Language

There is more to a data entry program than can be gleaned from a schema. This additional information sets the style of the program. This additional information needs to be passed to the form creation process. It can be saved as a resource description.

A GUI program can be used to define this style data.

Status: Planning.

3.3.3 Static Form Generator

The static forms are supplied as part of **SASSY**, and are built from the RDF schemas.

A plug-in for Gayal could load a description of the schema into a model based on the Entry Form Language (EFL) schema. The resulting model can then be used to generate static QML (Qt's GUI markup language). The back-end that connects the GUI to the RDF model might also be generated.

Status: Planning.

3.3.4 Dynamic Form Generator

These are for data entry forms for user created schemas. Rather than being part of the build, the process has to be run by the user.

The Gayal program and its EFL plug-in can create an RDF model of the form. This can then be used to generate dynamic QML. This dynamic QML will then need to be bound to an interpreter that interfaces to the RDF database.

Status: Planning.

3.3.5 Mind Mapper

This is for doing initial discovery, prior to having any details about the data.

The idea is to have a mind mapping style of program that is in the form of a graph. It will likely get quite large so it will need to support search and a variety of graph visualisation techniques.

It will also have hooks to schema based entry forms for various types of data. These would include a glossary of domain specific terminology, a bibliography, and a directory of people and organisations associated with the project..

Status: Seems like a good idea.

3.3.6 Document Generator

The document generator uses an RDF schema that describes the Markdown language. It is responsible for converting RDF data conforming to that schema into Markdown text.

Status: Working version. Needs rewrite to use *Gayal*.

Pandoc The markdown text is passed to Pandoc for conversion into a PDF document.

This conversion uses a LaTex template. It is expected that there will be a variety of templates that the user can select from, and, of course, modify for their project.

Status: Stable product.

3.3.7 Diagram Generator

Manually creating, an especially updating, diagrams is a fools errand. In practice the diagrams are either at such a high level as to be useless, or are so cluttered they are unreadable on anything but A0 size paper. They are, almost by definition, out of date.

The solution is to automatically create many small diagrams from the data in the knowledge database (not from diagram specific data).

Status: Seems like a good idea.

Diagram Description Language Diagrams can have a wide variety of styles and other similar attributes. A language for specifying these attributes seems like a way for the user to specify what is required, either by selecting from prdefined descriptions, or crafting their own.

Status: Seems like a good idea.

Diagram View The first step is to create an RDF view that encompasses the data that needs to be displayed. This can use the RDF view mechanism described in the RDF Utilities section.

Layout The selected view is then examined to find the focus path. This is based on the shortest path with the largest semantic strength. The remainder of the data (not on the focus path) is then sorted according to the Rhetorical Structure Theory algorithm.

The *ELK* tool can then be used to layout the nodes of the diagram.

Generation The layout data and the selected style information is then combined into an RDF model. This model is then parsed and used to create SVG for the diagram.

3.3.8 Table Planner

Presenting data in tabular form allows the reader to compare values in various ways that are not easy in other formats. **SASSY** documents need to be able to generate tables for various data sets.

Table Recogniser RDF has built in structures for sequences and lists. Such structures are a good indicator that a table might be useful. An RDF class that has numeric values or short strings can also be presented as a table if there are more than a few instances in a particular view.

Column Ordering One criteria for ordering the columns might be variability in the data in each column.

If there are object property relationships between the columns then the semantic strength can be used to place the more important columns first.

Status: Seems like a good idea.

3.3.9 View Planner

A requirement for **SASSY** is to be able to present the design information from different points of view. For example, the network engineer will need a different view than the database administrator. A mechanism is required that will select all the data applicable to a nominated view point.

A templated approach, or anything similar that tries to provide a detailed specification of what to present, is likely to leave out aspects of the design which are important for a specific project. We cannot hope to predict everything for all projects.

Scope The approach is to use a few anchor points in the knowledge database, determine a path between them, and then expand the area around that path to capture everything that might be relevant.

Content This module will be responsible for identifying tables, diagrams, paragraphs, and the presentation style for the selected data.

Status: Seems like a good idea.

3.3.10 English Text Planner

The module is responsible for converting the RDF data selected for a paragraph by the view planner into grammar trees for a set of sentences.

In addition to the data the planner will have a model for the reader so that the appropriate language is used, a model of the document for style information, and the context within the final document.

Focus Path The main focus path will be determined by finding the shortest path with the largest semantic strength. Each predicate in the RDF data has a semantic strength value assigned to it.

Rhetorical Structure Theory An algorithm has been devised that sorts nodes into a tree structure which when traversed provides a coherent sequence. The semantic strength of the predicates is used as the sorting criteria.

Sentence Delineation This is responsible for aggregating the RDF statements into sentences, or the clauses in lists, for example. It will try to create appropriately complex sentences according to the type of document and the profile of the intended audience.

Sentence Organisation This is responsible for things like setting the subject of a clause, and the relative subordination of the clauses.

Reference Choice This is responsible for replacing nouns with pronouns or pronominal phrases.

Lexical Choice This is responsible for word selection based on the reader and the context of the paragraph.

Grammar Generation This is responsible for generating the grammar tree for each sentence, passing it to the Natural Language Generator and assembling the results into a text paragraph.

Status: Under development.

3.3.11 Natural Language Generator

The Natural Language Generator (NLG) is a C++ rewrite of the Java program Simple Natural Language Generator.

The Java program has the functionality required, mostly, but tortures the Java language so badly that I cannot bare to look at it. Hence a rewrite in C++ using proper object oriented principles.

The Java program is advertised as being easy to convert to other languages. I have no direct evidence, but I suspect such a conversion would be quite difficult in practice. The basis for this claim is, I believe, the generic data structure underlying the program. This structure provides no constraints on the code and thus could handle any language, but at the cost of having to provide a whole mess of language specific code to manage the data. The C++ version abandons this data structure and reverts to a more conventional approach.

Missing from the Java program was the ability to set various orthographic features, such as italic and bold fonts, parenthesised phrases, etc. It is hoped that these can be introduced in a future version of NLG.

Status: Needs enhancement for orthographic features.

3.4 Processing

One of the goals of **SASSY** is to provide some automation to the development of the software architecture. This can be provided by dedicated programs or scripted with the rule engine.

3.5 Control

Many tasks that **SASSY** undertakes involve the coordinated operation of multiple programs. A declarative language will enable the user to define what needs to be done.

Part III

4 Ontologies

The ontologies are divided into two categories - those that are reference data (provided as part of **SASSY**) and those that record the details of the project being undertaken.

For this iteration of **SASSY** the ontologies are stored using RDF.

4.1 Schemas

These are provided as RDF/XML files, except where noted below.

Detailed reports of schemas can be generated by the *Gayal* program.

4.1.1 Analysis Schema

A core for the preliminary analysis mind mapping program. This will be copied from an RDF/XML file to a *PostgreSQL* database where it will be extended with the domain classes and properties.

4.1.2 Architectural Decision Record Schema

This schema defines the part of the knowledge database that holds the reasoning that went into the design.

4.1.3 Bibliography Schema

This schema defines that data that is recorded about documents that are referenced in the analysis and design.

4.1.4 Core Schema

The core schema is basically the well known elements of RDF, plus a few additions to smooth over a few omissions.

4.1.5 COTS Schema

The COTS schema defines that data that is to be recorded about products that were considered for inclusion into the project.

4.1.6 Diagram Description Schema

This schema defines the attributes that might be used to set the style of a diagram.

4.1.7 Directory Schema

A directory records the contact details and hierarchy of the people and organisations associated with the project.

4.1.8 Document Schema

The document schema defines a document that is ready to be converted into markdown text.

4.1.9 Entry Form Schema

This contains the specifications for the data that is needed to generate a data entry form, beyond that provided by the schema for the data.

4.1.10 GIS Schema

The geographic information system schema defines the data that is required to locate physical objects.

4.1.11 Glossary Schema

This defines the glossary. It provides domain specific meanings for terms as is normal for a glossary. It also provides a human friendly version of the RDF URIs. It also provides some data for managing the transformation of RDF into other formats, such as the semantic strength of predicates and the parts of speech.

4.1.12 Library Schema

The library schema defines the data required to identify and locate the documents, diagrams, images, and other media that has been collected or generated by the project.

4.1.13 Modelling Schema

A schema that specifies the data that can record UML (or similar) models.

4.1.14 Requirements Schema

This specifies that data that makes up the requirement specification.

4.1.15 Rule Schema

The rule schema defines rules for the rule engine.

4.1.16 Scheduling Schema

The scheduling data, defined by this schema, will start with the style, then broadened to show the build dependencies between modules. From there the tasks will be added, followed by estimates of the effort and duration.

4.1.17 System Design Schema

A schema for expressing the high level design of a system. This includes the module structure, interfaces, data sets and protocols, and the data transformations required. It must also include the physical design that relates programs to hardware.

4.1.18 Tactics Schema

This schema covers architectural design patterns, tactics and their responsibilities.

4.1.19 View Schema

Rather than specifying the data in a view, this schema while identify the data required for gathering the data for the view.

4.2 Reference Ontologies

These are typically provided as RDF/XML files. The main exception is the lexicon which is constructed from online sources during installation of **SASSY**. The lexicon is normally stored in a *PostgreSQL* database.

4.2.1 Lexicon

This is the union of *en.Wiktionary*, *WordNet*, and *Roget's Thesaurus*.

4.2.2 Quality Attributes

A collection of quality attributes as proposed by several researchers and an ISO standard.

4.2.3 Standard Tactics

A collection of well known architectural design patterns, tactics for addressing quality requirements, and the responsibilities associated with those tactics.

4.2.4 Preferred COTS

Some organisations have preferred products that are well supported and understood by the developers. This data set contains descriptions and attributes of these products.

Where applicable the description should include ratings for quality attributes, and compatibility with the standard tactics.

4.2.5 Data Entry Form Styles

Some default styles for data entry forms.

4.2.6 Diagram Styles

Default styles for diagrams.

4.3 Project Ontologies

These are typically held in a *PostgreSQL* database. This facilitates multi-user operation.

4.3.1 Catalogue

The catalogue holds the metadata for the other RDF models. This includes the type of storage and data for accessing the model, prefixes used as aliases for the URIs in each model, and submodels that are to be included.

The catalogue will be an RDF/XML file whose location is specified by the configuration file for the project.

4.3.2 Analysis

This model supports the preliminary analysis mind mapping process. Since its structure and contents are entirely project specific there isn't much that can be said about it.

4.3.3 Glossary

In addition to the usual definition of domain specific terms the glossary also has a mapping between URIs and English text labels.

4.3.4 Directory

A repository for contact details for the people and organisations that have some interaction with the project.

4.3.5 Library Catalogue

Most large projects will accumulate various documents, images, and other material that will be kept either on-line or in physical storage. This material needs to be located and cross referenced, hence a catalogue is required.

4.3.6 Bibliography

Enable the locating of documents that are referenced in the project documents.

4.3.7 GIS

The geographic location of the hardware components is an important part of a distributed system.

4.3.8 Existing System Model

For many projects the first step is to fully document the existing system so that essential functionality is not overlooked. This repository stores that information.

4.3.9 Requirements

This is the repository for the requirements for the project. It will include a unique identifier, a text description, indicators for importance and urgency, and links to the preliminary analysis. Large projects might include the domain if there are more than one.

The quality requirements should include links into the Quality Attribute repository. These will be used to guide the selection of tactics to use.

Domain specific terms should also have links to the entry in the glossary.

4.3.10 Schedule

Initially the schedule will only have the scheduling style since this can influence the tactics used in the solution. As development proceeds additional module dependency information can be included, and later estimates of the effort required.

4.3.11 Tactics

This repository will be initialised from the standard tactics based on the tactics related to the quality requirements that are mentioned in the requirements.

The set will be pared down to achieve the best overall match for the requirements.

4.3.12 COTS

All the off-the-shelf products that were considered should be described.

4.3.13 ADR

Architectural decision records should be created for each change to the set of tactics; for each decision about the applicability of a COTS product; and for each step of the system design process.

4.3.14 System Design

The system design data will start off with the module decomposition structure. This will be followed by the interface definitions and the data transformations. It will have links to tactics and requirements and the ADRs so it can be automatically checked for completeness.

4.4 Temporary Models

In addition to the above there will be numerous temporary models that are used to communicate between the various processes.