SASSY Background and Overview

March 2022

1 Introduction

1.1 The Development of Large Projects

The discussion will be primarily about software, but I think that many other areas have similar issues.

My IT career was cursed by monster projects. It started[1] with one that was in the billion dollar range back in the 1980's and never really improved. Unfortunately for this discussion most of the projects were things that I cannot write about in detail. So, now that I have retired, I thought I might try to develop something that would make it a bit easier to design these huge systems. I call the project SASSY for "Software Architecture Support SYstem".

My definition for a large project is one that needs more than one team of about five or is going to take more than about six months. It will also involve the creation of something that has not been built before. The almost inevitable consequence of those requirements is that there will be a need to create documentation so that the teams know what each other are doing and the reliance on human memory is reduced.

One of the common features of large projects, especially those for or by governments, is that they fail. Sometimes spectacularly, sometimes they are just declared a success anyway, but it seems that the bigger they are the less likely they are to make their users and clients happy. I think the underlying reason for most failures is that the project's architecture was deficient. This then causes the rest of the project to struggle until they either deliver something less than expected or the project is killed by its sponsors.

The observation that small projects are vastly more successful than large projects has lead some to simply say "No" to large projects and just do small ones. This doesn't fly well when dealing with CEOs and Government Ministers who have been sold the idea that they can save lots of money by investing in some gargantuan piece of software.

[1] Actually my first IT job was to help develop software for an aircraft carrier. If you know about the Australian Navy you will see the flaw in that project. (There isn't any such object.)

1.2 So, What is "Software Architecture"?

It is all the design decisions that need to be made before you get down to the detailed design and coding.

A large project will have a senior architect and will often have a team of specialists - hardware, network, database, etc. There will also be input from project management on the availability of staff and resources. Their decisions are all interdependent. Choices made in one aspect can affect the suitability of choices made elsewhere.

All of this design work tends to generate a huge amount of documentation. Keeping the published design up-to-date can be near impossible, and finding the data you need difficult as it's often embedded in details that are not relevant. This rather inevitably leads to some designs being made that ignore other parts of the design that the designer was not aware of. These only come to light during coding or even testing when it can be too late to fix.

This brings us to SASSY. Its goal is to store the design data in machine readable form (probably in a linked data database) so that tools can be used to find discrepancies, contradictions and omissions and produce reports showing the system from various viewpoints. It should also have the tools to assist in the design process.

1.3 SASSY Status

SASSY is, itself, a large project. My intention is to use it as its own test case so it can eventually assist with its own development. (I do like recursive solutions to problems.)

Currently it is in the feasibility study phase. There are many, many parts that need to be investigated, and a few of them may turn out to be too difficult. My main focus is on getting the data out in useful reports since without that the whole concept crashes and burns. So far I have built a C++ wrapper for the Redland RDF library and GUI program for viewing and editing RDF data. This is proving to be quite nice for building RDF based programs.

I have also been investigating how to turn RDF data into something like readable English. I took the natural language generator Java program, SimpleNLG, and rewrote it in C++ (mostly because I did not like the way they used Java). This can turn a grammar tree into a sentence and should be able to be easily extended for what SASSY needs. For the last few months I have studying the problem of how to turn a set of RDF statements into a paragraph of text. This is going to be complicated!

My approach to constructing a SASSY prototype is to start at the output end and try to work backwards up the processing stream. I know what the reports should look like so I wrote a few examples using a word processor as test cases. Next I found a program call PanDoc that can create PDF from Markdown and

used that to duplicate the test cases. This gave me the information I needed to design the preceding processing step. I have created an RDF schema for a report document and built a document in an RDF database. I have built a C++ program that can generate Markdown from RDF. After that I will look at building the RDF documents using rule engines, or something.

Sadly it is rather obvious that I will probably never see a finished version of SASSY (it would make a nice 150th birthday present, though.) Hence I am just picking off the interesting bits and seeing what can be done.

2 Software Architecture

I thought I might provide a bit of background by describing what software architecture (SA) is all about. But SA is not where the story starts, it is the middle of process. So lets go right back to the beginning...

First there was light, then there was a Big Bang. Oops, too far back.

2.1 Preliminary Analysis

Once a decision has been made to start a project the next step is to find out as much as possible about it. This step is usually called "Preliminary Analysis". If the project is familiar territory it might only take an hour or so. Or, it might take a team of dozens of analysts several years. I have seen the latter case too often, and it is not a good sign for a successful project.

Perhaps a more down to earth example might be illustrative: Imagine you have been asked to add a cabinet to the entertainment system. First you need to get a rough idea of what it is for - lets say storage of DVDs, CDs and other media. From this we see that it will need to be of good quality. So we check out furniture suppliers, since buying ready made is often the easiest and cheapest solution. If nothing is found we need to investigate types of wood, fasteners, hinges, glue, veneers, etc We may need to acquire some suitable tools.

In other words, we need to find out about what the project is all about.

Sometimes you will need to check the feasibility of the project. Perhaps it is going to require unobtainium, or require faster than light communications, or just take more money or resources than are available. Your research might have uncovered some bit of software that looks like it might be useful. It is a good idea to test it out to find if it is as good as they claim.

Most of the really big projects I was involved in were to replace some existing system that was reaching the end of its useful life. In such projects it is necessary to get a really good picture of the existing software. It annoys users no end when the shiny new replacement doesn't have half the functionality they were used to. Building a detailed model of an existing system can be an almost impossibly complex task.

At this point we have a pretty good idea of what is going to be involved. A useful task at this point is to build a data dictionary, or at least a glossary of terms. Systems, and the people using them, tend to develop their on specialist language. It is reassuring to the client if the development team uses the same language.

2.2 Requirements

It is time to sit down with the users and build a detailed set of requirements.

I like to divide the requirements into three sections:

- The first is the *functional requirements*. These are the things the user will ask you to provide. It is what the system must actually do.
- The second is the *environmental requirements*. If the project is being built for an organisation that is committed to Microsoft, then that is a requirement.
- The third is the *quality requirements*. This covers things like security, performance, etc (I have a list of about 100)

An interesting thing is that it is often the quality requirements that have the biggest influence on the architectural design.

We have now reached the point were we know what is wanted. We can start to think about how to deliver it.

2.3 Software Architecture Tactics

The components of a system have a set of responsibilities. Each component has assigned to it a set of things that it is responsible for.

For the functional requirements these normally map directly to a component's responsibilities. That is, a particular component will be responsible for a particular system function.

For the quality requirements however there is no single component that is directly responsible. No single component could be solely responsible for the performance of the system. A tactic is what maps these requirements to a set of responsibilities which can then be assigned to components. For example a scalability requirement might be met by using a client-server design – the tactic. We can then map the consequential responsibilities to components; in this case by putting a service in one component and a client in another, and perhaps a name lookup service in a third.

Thus the design process is one of selecting a set of tactics that give the best response to each of the quality requirements. Of course there will be competition since using one tactic might compromise the use of another – it is hard to have a system that has both good security and good usability, for example.

The first part of developing a solution for a set of requirements is to develop the tactics to be used.

2.3.1 Functional Requirements

For the functional requirements this typically means trying to find a set of existing software that will work together, satisfy as many requirements as possible, while not conflicting with the environmental and quality requirements. The gaps will have to be filled with software developed specifically for the project.

The process of selecting the software will have to take into account the costs and timing. Third party commercial software will usually involve licensing costs.

Open source avoids that but will often require an investment in time to determine if it has any shortcomings that will need to be fixed.

2.3.2 Environmental Requirements

The environmental requirements are usually used to rule out options, rather than require anything to be constructed. Of course it might be that too much is ruled out and you will have to either reconsider the requirement or commit to a lot more development.

2.3.3 Quality Requirements

The quality requirements (also commonly called the "non-functional requirements") are the important input for the design of the system architecture.

2.4 Software Architecture Components

Once the tactics have been selected you will have a corresponding set of responsibilities - the things that need to be done to fulfil those tactics.

The responsibilities then need to be assigned. Some (hopefully, most) will go to existing programs and libraries that will form part of the solution. Some will go to software that has to be created for the project. Some responsibilities go to the development process itself.

That last group may sound a little surprising so some examples:

If your tactic for getting good quality was to do code reviews then there is a responsibility for setting up the review process.

If your tactic for performance was a client-server architecture pattern there would be responsibilities for assigning processing to clients and servers.

Assigning a responsibility to some existing software should result in a test harness being constructed to confirm that the responsibility is fulfilled. You will need to keep that test harness for the lifetime of the project so that the maintenance team can verify that new versions of the component continue to work as required.

The remaining responsibilities will have to be assigned to components that you need to create. The process starts with the complete list and then partitions it into separate components. Large components should then be subdivided, until all components are at a size that can be handled by single team in a reasonable time.

There are various inputs to the subdividing process: You can use well known architectural design patterns, (such as client-server, or message queues). User interfaces can be guided by the use cases that the users suggested as the way the system will be used.

My experience with very large projects leads me to the conclusion that the first partitions should be on political lines. This has the advantage that it also divides the stakeholders which can be a significant step towards getting a solution. A variation on the political division idea is to also include a "technical" partition in addition to the functional partitions. This is made responsible for the common aspects of the system. It can be an easy sell to the stakeholders as it reduces their individual costs, being a shared component. The stakeholders will then also gain an appreciation of the cost of having a special component, and may decide that the cheaper alternative is to modify the business process instead.

An important constraint on the component subdivision process is the interfaces between the components. Keeping the number of interfaces on each module to a minimum, and also keeping them mostly to "adjacent" components will result in a simpler and more modular design. Assigning components to layers is another useful technique. You also need to keep an eye on the data flows - a component can only output information that it was able to construct from its inputs. "Magic happens here" is not a useful approach.

2.5 Software Architecture Interfaces

At this stage we have divided our monster project into a large number of component pieces that can each be developed by a small team in a reasonable amount of time. So we hire a herd of developers, assign them to the components and let them get to work on their subprojects. This involves doing a detailed design and the coding and unit testing. Another team will be responsible for high level testing.

Most of the components have interfaces to other components, and these interfaces are a major constraint on the design of each component. Since they have not been designed yet, each team has to make assumptions about the other modules. The data and the protocol for the interface will be defined in the architecture but that still leaves a lot of fine detail to be negotiated between the teams.

For the next few months great progress is made. The designs are done, coding is proceeding apace. It looks like its about 80% done. Then things seem to stop advancing. What is going wrong?

The teams are submitting their modules for inclusion into the system build but this is uncovering problems. Sometimes it is compile time problems as the modules have used out of date versions of other modules, sometimes it fails in testing due to different interpretations of interfaces. This causes the teams to make changes to their code, but often these changes cause other modules to fail.

2.5.1 A little maths.

If you have a system that is described by some variables, and where the rate of change of those variables depends on the value of the variables, then as time goes by it will adjust itself into a stable state. However if your rate of change

depends on what the values were a few moments earlier the system may no longer converge but instead it will oscillate erratically. This is known as "chaos".

The larger the delay the worse the oscillations become until eventually the system is wildly out of control.

2.5.2 Back to our project.

It seems our project is in a similar chaotic state. The developers are always working with out of date versions of the other modules, and this is leading to an endless cycle of changes. It is not going to converge to a working system. I believe that Microsoft got into this state when developing Vista.

One technique often used in this scenario is to do overnight, or even continuous builds of the system so things are not out of date. This probably helps, but is not a silver bullet since it is not the code that is the problem, but the coders. Their understanding of the other modules is what is out of date.

2.5.3 Independent Subprojects

While we have divided the system into small components we still have a monster project. The components all interact with each other. What we need to do is divide the monster project into small independent projects.

This can be done by taking each interface between pairs of components and building two test harnesses, each emulating one of the components. Each test harness will have two parts, one that initiates tests and one that responds to requests.

As a side benefit the test harnesses can be used to test the interfaces themselves. Some interfaces will involve interprocess communications such as pipes, sockets, message queues, through to web services. It is nice to know that they work properly.

We can now develop each component in complete isolation. Each one is an independent little project. All it has to do is fulfil the responsibilities assigned to it and pass the tests from the interface test harnesses.

I confirmed that this approach can be implemented and works in the VICI project.

2.5.4 Layers

While the above independent project approach would work it is not very satisfying for the developers. For example, if one of the other dependent projects was a GUI interface, you would end up developing GUI programs without actually seeing them in operation.

This can be addressed by developing in layers. Assign teams to work on projects at the bottom layer first, and work up through the layers.

2.6 Software Architecture Scheduling

It is often the case when building very large systems that you will need to construct quite a bit of infrastructure before you can start delivering the functions that the users can use. This can cause some unease in management as they see the IT budget being chewed up by a large team of developers and nothing useful appearing.

The layered approach that was suggested above only makes this situation worse. All the initial effort goes into developing the libraries for the low level components, and nothing for the users.

Using some pre-existing frameworks is one way to avoid the "nothing for the users" problem. You can often start delivering some functionality quite quickly. This is why frameworks are popular. However, let's assume that this is not an option for your project.

One way forward is to lash up some temporary solutions for the infrastructure components. The user functions can then be delivered and sort of work - perhaps not very quickly or reliably or only for very few users. I would strongly recommend that the first functions delivered be provided "for demonstration purposes only" and be used to show that progress is being made. The danger is that the temporary solutions might be judged as the quality of what is to come, which could lose you some support.

The biggest danger is starting with a simple architecture and trying to evolve it into something more complex and fit for purpose. Trying to change the architectural design almost never ends well. You will be forever trying to clean up the stuff that was built to the earlier design. (Have a look at the Ariane 5 failure for an expensive example.)

The implication is that you have to build the architecture based on the full set of requirements. You cannot select a subset of requirements, develop a design, and then add more requirements later. Software designs are not smoothly related to the set of requirements. While small changes can often be accommodated (otherwise maintenance would be impossible) there always comes a point where a seemingly small change should result in a major redesign.

Hence, part of your architectural design will be the development schedule. Its aim will be to provide functionality as soon as possible. Some components will have to have temporary solutions, perhaps using scripts rather than compiled code, for example, or simple text files instead of database access. It would probably be a good idea to put something in place to prevent temporary solutions from becoming permanent - we have all seen that happen too often.

2.6.1 Summary

I have, in this section, tried to describe the environment in which the SASSY system will operate.

SASSY's purpose is to automate the routine tasks and guide the architects into building an optimal design for their system.

3 SASSY, An Overview

In the previous section I tried to give a rough outline of what happens during the early phases of a project. Next I would like to explore what the SASSY software might be able to help us with.

3.1 Preliminary Tasks

3.1.1 Preliminary Analysis

I think the earlier that data gets captured by SASSY the better. Otherwise it will be captured into other formats, such as word processor documents, and will need to be transcribed into SASSY later. Hence, it would be useful if we included something like a Mind Mapping module that can be used to capture information about the system to be developed as it is discovered.

This leads to the need to import the mind map data into SASSY's database, and also to present it in a nice readable format. I will return to the presentation aspect of SASSY later.

For the SASSY prototype I am considering using VYM (View Your Mind).

3.1.2 Modelling

For many systems it will be necessary to capture the details of existing systems that are going to be replaced. A module for recording the data structures and processing would be quite useful.

For the SASSY prototype the UML module of Dia should suffice.

3.1.3 Data Dictionary

This will start out as a glossary of project specific terms and abbreviations. As the project progresses it will include data types and formats, valid ranges, and so forth. This will be most important for the interfaces between the subprojects. It should include the ability to cross reference the entries.

Data entry will be via a SASSY GUI.

3.1.4 Project Schema

This will allow the user to create a schema for their project. It shows class relationships, data properties, and the relationships between objects.

Data entry will be via a SASSY GUI. (This GUI already exists as part of the RDFGUI program.)

3.1.5 Requirements

Initially this will be a textual list of the requirements. Each will be given an identifier and a priority. Later it will also have the development phase in which it is expected to be delivered.

It would be good to be able to express the requirements in RDF so that they can be analysed. Something that needs to be prototyped, I guess.

This will also use a SASSY GUI for data entry.

3.1.6 Quality Attributes

The requirements that impact the quality of the system are a fairly well known set. SASSY will provide a reference library of these quality attributes that can be used as a check list when formulating them for the project.

3.2 Architectural Design

With the preliminaries done and a clear idea of what the proposed software should do it is time to start the design process.

3.2.1 Functional Tactics

The first step is to select the existing software that can be used to satisfy the functional requirements. Some organisations may have preferred applications that should be used. SASSY will provide its administrators the ability to list these programs. It would be nice if it had a full list of all available software to select from, but that seems like too much for now. (Perhaps links into Wikipedia might work.)

Any requirements that cannot be met by existing software will be associated with the software development tactic.

3.2.2 Environmental Tactics

These are usually expressed as constraints or things that need to be done or avoided. The tactics are usually something like having procedures in place to ensure the constraints are observed during the development process.

3.2.3 Quality Tactics

These are the tactics that satisfy the non-functional requirements. Things like performance, number of users supported, security, and so on are the focus. The tactics chosen will greatly impact the overall design of the system, and how well the result performs.

SASSY's library of tactics will include information on how they interact with each other. The architects will be expected to describe why they made the choices they did. This will be important if the system ever needs major enhancements.

SASSY will have the ability to report on the association between requirements and tactics. This will highlight anything that might have been overlooked.

3.2.4 Responsibilities

The tactics chosen each have a set of responsibilities that must be satisfied if the project is to meet its requirements. For the Quality Tactics these will be provided by SASSY as they are well defined. For the Functional Tactics the architect will need to add responsibilities corresponding to the functional requirements.

Responsibilities can be classified into two groups: Those that are satisfied by the developed software, and those that are satisfied by the development process itself. For example, an environmental requirement that the project must use Open Source software would result in a tactic of monitoring the software selections to ensure open source is used. This would then have a set of responsibilities for reviewing the software selection process - any task that involved software selection would have an associated procedure for doing that review.

The architect will need to select the responsibilities that relate to the development process and assign them to the development tasks. SASSY will have a default set of development tasks, but a project can modify it for their own development process.

3.2.5 Components

All the responsibilities that are to be satisfied by the delivered software need to be allocated to components that are to be developed. The tactics will have associated design patterns that can provide an initial set of components.

3.2.6 Conceptual Design

The initial set is likely to have just a few components each with a multitude of responsibilities. The architect's job now is to subdivide the components until each one can be developed by a small team in a reasonable amount of time. Each component should be like a mini-project that makes sense in isolation. It should have a single purpose or theme. It may even be necessary to split some responsibilities so that the components are small enough.

SASSY should include tools to define the components and assign their responsibilities. It should be able to produce reports describing the system, including how the requirements are going to be met by the components.

3.2.7 Scheduling

By reviewing the components against the importance and urgency of the requirements a development schedule can be created. This may involve introducing some temporary versions for some components if that can deliver functionality for early versions of the system.

Note: Previously I have just assigned a priority to requirements. These have values from mandatory through to "nice to have". It occurs to me that such values only apply to the finished design or system. In order to schedule the development of a system that will be delivered in phases it would be useful to also have urgency as an attribute of each requirement. It may be that some things will be important to the final acceptance but not necessary for the initial phases.

3.2.8 Logical Design

In this step the components are allocated to runtime modules, such as libraries and programs. The interfaces between the components are defined.

SASSY should provide tools to assist with the interface design. For example it should be able to confirm that the data flowing out of each component has a properly defined source.

3.2.9 Interfaces

This is one of the more important outputs of the design process. If the interfaces are well defined and correct it becomes possible for each component to be built and tested in isolation. Test harnesses that simulate the connected components allow the component to be built with high confidence that it will operate correctly in the final system.

3.2.10 Testing

The logical design is enough to allow the test team to start building test harnesses for the modules. The test harnesses should, obviously, test the functionality, but they should also test the quality (such as performance) of the system as well.

Each interface can have a set of test harnesses and simulators built. Initially this can test the interface itself (important for things like web services) and later they can provide a test environment for the modules so they do not have to rely on other modules.

SASSY should have enough details of the interfaces and the data involved to be able to provide a lot of the design information for these test harnesses. Perhaps it might eventually be able to generate them.

3.2.11 Physical Design

In this step the modules (eg libraries and programs) are assigned to physical hardware or virtual machines. It will also involve things like network equipment, and any other hardware required. Things like load balancing, firewalls, isolated environments, distributed databases are designed.

The physical design should also provide a design for the development and testing environments. A typical large project will have separate environments for

programming, unit testing, system testing, acceptance testing and production. Each will need a set of hardware that suits the tasks.

3.3 Application Development

We have now reached the end of what SASSY is intended to cover. The design documents are passed to the development and testing teams, the network engineers, database administrators, and project managers.

It is, however, easy to imagine SASSY being extended into the application detailed design process.

3.4 Documentation

SASSY has four roles to play:

- Provide a library of software architecture information, such as quality attributes, design patterns, etc.
- Provide the ability to enter a description of the system to be developed and the decisions made during the design process.
- Provide tools to check the consistency of the design, check for omissions, etc.
- Provide reports that give the developers a clear and unambiguous view of the design from a particular point of view.

It was the reporting aspect that provided the incentive to start the SASSY project. Watching large projects create multiple conflicting documents, or documents where the required information was intermixed with other irrelevant information, or failing to have any design documents at all was embarrassing.

Most projects end up using word processors for their design documentation. This is manually intensive which is another way of saying it isn't done in a timely fashion. When it is done it is often out of date before it gets to the people who need it, or it is written from multiple points of view at once resulting in confusion.

4 Document Generation

If we define a "bug" to be a software that does not do what it was designed to do, we can immediately see why the claim "The code is the design." is deeply flawed. All programs would be bug free by definition! The problem is that the code perfectly describes what the program does, but provides very little information on what it was intended to do.

When developing a large system that is going to take a lot of people a lot of time then the design documentation is all we have for them to communicate the **intentions** of the software.

The problem is that we are very poor at creating these design documents. They are often not kept up to date - there can easily be multiple versions, with significant differences, in circulation. They are frequently written from multiple points of view which results in a confusing mess where it can be difficult to find important information.

4.1 Requirement

One of SASSY's goals is to make the design documents available whenever needed and in a form tailored for the specific point of view. For example requesting a document with a network point of view will create a document specifically for the network engineers.

4.2 Tactic

In order to satisfy this requirement the tactic is to generate the document when requested directly from the design information held in a central knowledge database.

The SASSY knowledge database has two categories of data: There is a set of reference databases that have a format that will be known to the developers of SASSY. Creating reports from this data will be much the same as any other report program - fill in a template with the results of predefined queries.

The second category is the data for the project SASSY is being used to design. Every project is different and will have its own unique data sets. The template and predefined query approach will not work. If the data set contained data that the query system was not expecting it would be missed - and this is exactly the information that would be important for the developers.

I am still researching how to do this, but it seems there will be three parts:

- A discourse module that will explore regions of the knowledge database and create a high level plan.
- A microplanner module that will organise a small set of RDF statements into a paragraph.

- A Natural Language Generator that will turn the sentences of the paragraph into English text. (This bit is done, mostly.)

The results of the request will be collected into an RDF representation of a document. This is then converted to Markdown text which is then passed to the PanDoc program to create the required format output, typicaly PDF. A prototype for this has been created.

I must admit that I had a vague hope that there would be existing microplanners that I might use or adapt. I was a bit surprised to find that it is still an academic research subject that has not made much progress in the last 40 years. Hence I have some doubts about the viability of this approach. This is what feasibility studies are for.

A lot of the academic work was done trying to put the RDF statements (or the equivalent) into a coherent order for a paragraph. All the solutions I could find were either in extremely limited domains or used algorithms that were exponential in performance - effectively limited to about 8 or 10 RDF statements.

The approach in the literature was to devise a plan, often with complex mechanisms to make it flexible and then try to find data in the knowledge database that would satisfy the plan. This seemed odd to me - I usually start with the knowledge that I want to communicate and then devise a plan to express it. I have built a small test that seems to work quite well using this approach.

4.3 Text Generation

This is the bit of SASSY that is least likely to be successful. Generating text from the contents of a database is something that people have been trying to do, probably since the first Teletype machine was connected to a computer. The lack of readily available software for this task after all this time is a good clue that it is not going to be an easy problem.

The problem can be subdivided into three parts:

- A Discourse Planner that explores the knowledge database to find the data that relates to the topic requested. I have some ideas on how to do this using a bit of graph theory and rule engines.
- A Microplanner that converts a small set of RDF statements into a set of sentences (aka paragraph) in the form of grammar trees. This is the heart of the problem that I will describe in more detail below.
- A Natural Language Generator (NLG) that converts the grammar trees into English text. I will go into the details of this in a future post where I go deep into the individual components.

4.4 Microplanner

The inputs will be a set of RDF statements that we need to arrange into a paragraph of text. The planner will also have information on the intended reader, the type of document being created, and where in the document the paragraph will appear. This extra information can be used to make style and content decisions.

The output is a set of grammar trees suitable for the NLG.

4.4.1 Concepts

The nodes of an RDF statement are URIs (things that look like web addresses) which are, obviously, not what we want in our report. Each one needs to be associated with a concept which is one or more words that have the meaning intended by the node. It is good practice when building an RDF model to attach labels to things so most nodes should have something we can use as a concept. Often it is also possible to extract a suitable word from the URI itself.

These labels then need to be given a more solid meaning and classified according to the part of speech. I think this can be done by linking to entries from WordNet and enWiktionary.

4.4.2 Discourse Structure

This is responsible for sorting the RDF statements into a coherent order. I have read many, many academic papers on this topic and it seems that no one has devised a good solution. If you are interested look up papers on Rhetorical Structure Theory (RST). While RST would appear to be important for producing coherent text there wasn't any practical implementations described.

The approach chosen by most researchers was to devise a set of plan objects that reflected the communicative goal and then try to find data in the knowledge database to satisfy the plans. From SASSY's perspective this wasn't very good as there was no guarantee that all the relevent data would find its way into the report. One researcher identified this problem and tried to develop an algorithm for sorting the data into a tree structure that conformed to RST. This was not very useful as it was exponential in running time - each new RDF statement multiplied the run time by the number of statements - so 10 was about the limit.

I thought that the idea of starting with a plan was the problem. When writing something I start with the data I want to present and then devise a suitable plan to express it. A small experiment seems to indicate that this might work, but I will need to do a lot more testing to be sure.

4.4.3 Sentence Deliniation

This bit is responsible for placing the RDF statements (which are now in a coherent sequence) into sentence sized groups. A simple approach would be

to group together statements that share a common subject node. Some of the academic papers do provide useful guidance on how to use RST to create the sentence groups.

4.4.4 Reference Choice

This bit is responsible for replacing noun phrases (eg "John") with pronouns (eg "him") and definite noun phrases (eg "the boy"). I am currently reviewing a 250 page paper on this subject - it is surprisingly complicated. The difficulty is to avoid replacements that the reader will not be able to resolve.

4.4.5 Sentence Organisation

This bit is responsible for things like conjunctions, adjectives, subordinate clauses, and so on. It will also set up the grammar flags, such as tense and person, that the NLG requires.

I have always had a problem with tense in design documents. Until the code has been written the future tense seems more appropriate, but once its done it no longer feels right. With SASSY we should be able to just flip a bit in the document request and have it all change from future to past tense.

4.4.6 Lexical Choice

In a general purpose text generation program the concept words can be replaced with other words depending on the intended reader and the type of document. It is also possible that the concept words can be replaced by words from another language.

I am not sure this is necessary for SASSY.

4.5 Summary

I am still researching the microplanner, but I think the solution will be along the lines described above. I think this about covers the basic concepts for SASSY that I have so far investigated. I will probably come back later and go into the details of those parts which have working prototypes.

5 SASSY Development

5.1 Current Status

My aim is to build a prototype version of SASSY and then use that to do the architectural design for the real version. The prototype will also serve as a feasibility study for those components of the system that don't have obvious or straight forward solutions.

Currently the following components exist, though they need further work.

5.1.1 Infrastructure

I have a pair of infrastructure libraries that I have been building on for many years. One is for stuff that goes into most programs, and includes things like a configuration file handler, XML support, child process management, string functions, exception objects, etc. The the other is for testing. It has a test harness that can handle multi-threaded asynchronous processing. These have been slightly adapted for SASSY.

5.1.2 RDF Library

I found a C++ library that was a wrapper for the Redland C RDF library. On examination it needed a bit of work. It is now a fully functional C++ wrapper, using C++ paradigms and completely hiding the C code. The Redland library seems to have some short comings when working with large complex models, but I have added some code to allow multiple RDF models (a.k.a databases) to be combined. There is also support for a catalogue of models which is very useful for large projects.

5.1.3 RDF GUI

In order to try out RDF ideas I needed a program with a GUI interface. This has been thrown together without much thought to an overall design. It works, but the occasional crash can be expected. The program has tabs for editing RDF triples, viewing models, and constructing schema. It also has a forms interface that automatically constructs data entry forms based on the schema. This is better than nothing but a better approach is required for SASSY.

5.1.4 Reasoner

A Reasoner is able to check an RDF model for inconsistencies and infer additional relationships. This will be a useful tool for the users of SASSY as they develop their projects.

An RDF Reasoner has been built around the FaCT++ reasoner. It works but needs to better handle inconsistencies. (This is a common problem with reasoners.)

5.2 Natural Language Generator

This started out as a Java program called SimpleNLG (developed by University of Aberdeen). While it contained the rules for constructing English sentences the code was not something I ever wanted in one of my projects. Hence SASSY now has an NLG module written in C++. The original XML interface has been replaced with an RDF interface.

5.2.1 Document Generation

The Pandoc program provides the final stage in document generation. At the moment it has been set up to create PDF from Markdown input.

A C++ program has been written which creates suitable Markdown from an RDF description of a document.

5.3 The Path Ahead

The next step in the prototype is to build something which creates the RDF description of a document from some well defined RDF models, such as reference data or a schema. However, this will require a few other things first:

5.3.1 Rule Engine

The NLG module used a simple rule engine as part of its RDF interface. This needs to be generalised into a stand-alone component. Since building an RDF document from a schema is going to require a lot of rules there will also need to be some nice way of entering and editing them.

5.3.2 Prolog

The SWI-Prolog system has excellent support for RDF, and building the rules in Prolog might be a way to do this. An interface between SASSY and SWI-Prolog will need to be constructed. At some point I am going to have to learn how to use Prolog.

5.3.3 Templates

The first documents to be generated will be reporting on schemas and reference data. These have well known structures so the template approach is applicable. The template's slots will be filled in from queries on the knowledge database. Associated with each query will be a set of instructions on how to arrange its results into the template slot. The combination of a query and these instructions is called a rule.