Developing a Requirements Management Toolset: Lessons Learned

Muhammad Ali Babar [a], Didar Zowghi [b]
[a] *National ICT Australia Ltd. and University of New South Wales, Australia*, [b] *University of Technology – Sydney*
malibaba@cse.unsw.edu.au, didar@it.uts.edu.au

Abstract

Requirements Engineering (RE) is a multi-faceted discipline involving various methods, techniques and tools. RE researchers and practitioners are emphasizing the importance of having an integrated RE process. The need for an integrated toolset to support the effective management of such an integrated RE process cannot be over-emphasized. Tools integration has been identified as an important next step toward the future of requirements management tools. This paper reports on some of the significant architectural and technical issues encountered and the lessons learned in the process of developing an integrated Requirements Management (RM) Toolset: PARsed Natural language Input Processor (PARSNIP) by integrating various independent tools. This paper provides insights on architectural and technological issues typical of these types of projects, the approaches and techniques used to address the architectural mismatches and the technological incompatibilities.

1. Introduction

Requirements engineering (RE) is the process that incorporates all the activities required to identify, analyze, document, and manage the requirements of a software-based system. RE is considered one of the most complex and difficult activities and any deficiency in this process may lead to project failures [19]. It is a multi-faceted discipline involving various methods, techniques and tools; and requirements engineers are expected to have a wide variety of skills drawing upon a number of disciplines. To enable effective management of such a core and challenging process, the importance of automated support has long been realized [27-28]. Such tools are supposed to facilitate the process of managing functional and non-functional requirements of large and complex software systems. Automated support for the RE process can be of great benefit to a requirements engineer in successfully performing a number of tasks throughout the software development lifecycle; e.g. requirements management, requirements structuring, consistency checking, managing requirements creep, traceability, and so forth [6, 17]. Commercial and research organizations have developed a number of tools (For example, RequisitePro, Reconcile, and DOORS to name a few) that can assist in the various tasks of the RE process.

RE researchers and practitioners have also been pressing the need for an integrated RE process. It has been argued that instead of applying different RE methods, techniques and activities during various development stages, an integrated approach to manage the RE process is essential to gain the true benefits of applying engineering principles to this domain [20]. The RE community is not only working continuously on developing and validating various approaches to manage and integrate a variety of RE methods, techniques and activities in a coherent fashion, but it is also pressing the need for an integrated toolset to enable the effective management of an integrated RE process. Such a toolset needs to incorporate as many of the desirable functionalities provided by individual RE tools as possible [1]. A user of such an integrated toolset should be able to interact with the environment through a uniform interface without facing each tool's idiosyncratic interface. An integrated toolset also eliminates the need to keep the same or similar information in multiple repositories and the need to provide the same information multiple times to different individual tools. Moreover, when there are so many tools available, the development of an integrated support environment by integrating existing tools is a more plausible choice [13, 30].

We undertook a research and development project to investigate the architectural issues, approaches, and techniques required to successfully develop a RM

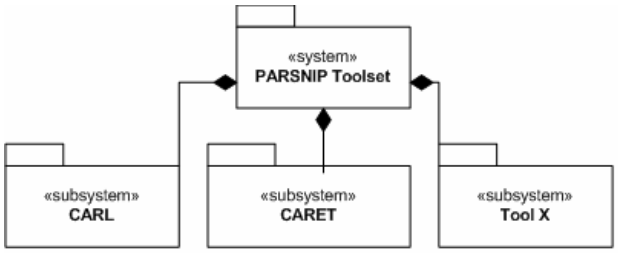

Figure 1The PARSNIP system as composed of various subsystems [22]

toolset by integrating a number of stand-alone RE tools. The aim of the project was to design and develop an integrated toolset that not only tightly integrates a number of stand-alone systems developed to provide a number of functionalities to support RE process (e.g., CARL, CARET etc.) but is also flexible and extensible enough to support future enhancements. The available standalone systems were fully functional and capable of performing their respective functions independently. To support the RE process in a coherent fashion, these systems needed to be assembled into an integrated RM toolset that could easily be maintained and enhanced.

This paper presents the insights we gained while developing a component-based RM toolset. The paper is organized as follows: section 2 provides a brief overview of the toolset and its three components. Section 3 talks about the architectural issues and our approach to addressing them. Section 4 explains PARSNIP's architecture. Section 5 describes integration approach. Section 6 presents the technological choices. Section 7 describes the lessons we learned and section 8 contains the conclusions.

2. An overview of the toolset and its components

Being an integrated RM toolset, PARSNIP provides a number of functions and features to support various activities of the RE process (such as domain modeling, requirements structuring, requirements management, consistency checking etc.) in a coherent fashion. To enable PARSNIP to support a wide range of RE activities, it consists of a number of independent or standalone tools. A high level view of PARSNIP has been shown in figure 1 using package notation with system and subsystem stereotypes of Unified Modeling Language (UML) [22].

This high level representation of the structural organization of PARSNIP shows that it can easily be decomposed into a collection of sub-systems. However, we have already mentioned that each of the sub-systems incorporated into PARSNIP is an independent system at different levels of abstraction. Like any other well-structured system, PARSNIP is functionally, logically, and physically a cohesive system, formed of loosely coupled subsystems. The aggregation relationship between PARSNIP and its sub-systems also demonstrates its extensibility by incorporating other systems, which are designed and developed to work as sub-systems if required.

The Computer-Assisted Requirements Evolution toolset, **CARET**, has been developed to provide a support environment to reason about inconsistencies during requirements evolution [7]. Requirements are entered as expressions in logical notation; each logical expression must have a priority (a numerical value) attached to it and it may also have a description of the requirement in natural language (English only). The algorithm for checking and reporting any inconsistency between requirements of a requirement set has been based on proven and well-known theories of classic logic, non-monotonic reasoning, and belief revision. CARET performs consistency checking on sets of requirements based on theoretical concepts, which have been extensively published in research literature [7] so our paper does not elaborate on any of these concepts.

The tool X[1] supports requirements specification and procurement process. It was developed by our industry partners**.** This tool provides a Graphical User Interface to structure and manage requirements, along with a single repository to store requirements at various stages of the software development process. It also provides process support in developing and managing requirement specifications and evaluating responses to them. The process guidance is embedded in its GUI and is reflected in its repository structure.

The Computer-Assisted Reasoning in natural Language (CARL) toolset was developed as a prototype tool to apply the CARET framework to natural language requirements and this work has been reported in [6]. This component was using Cico as its kernel for natural language parsing. Cico is a domain-based parser that uses shallow parsing techniques and exploits knowledge about domain specific properties

[1] Because of contractual constraints, we call it tool X.

[15, 16]. This natural language processing application was intended to provide an interface for entering a requirement in controlled natural language and translating that requirement into logical notation acceptable to CARET. However, there were a number of problems with this type of ad hoc arrangement between these applications, e.g. data had to be transferred between tools using text files and the parsing of compound nouns was problematical.

3. Architectural Issues and Approach

Software architecture (SA) is one of the initial and most important design artifacts in software development process. It has been shown that an architecture-based approach to developing software-based systems either from scratch or from existing components is quite effective in minimizing the project cost and increasing the quality of the end product [8]. Recently, the architecture-based approach has emerged as a successful means of developing component-based systems and managing complex system integration projects. Rigorous effort invested in architectural design and evaluation activities results in increased comprehension of the system, better communication among stakeholders, effective project management, controlled evolution, and rapid development [4, 10].

SA provides a high level view of various components of a system, connectors for the interaction of those components, and their topological description. SA exposes certain properties of a system, while hiding their implementation details. Such an abstract description of the system enables the development team to abstract away the irrelevant details and complexity and focus on the overall structure of the system. An architecture-based approach focuses on analyzing and evaluating the architecture of the existing component through architectural documentation or architectural archaeology. It identifies any mismatches amongst those components, Moreover, it encourages designing a high level architecture that can result in a system that fulfills functional and quality requirements, documenting, disseminating, and maintaining the architectural description, reasoning about the detailed design in the context of high level architecture, and so forth [3, 9, and 10].

Our initial architectural analysis of the candidate components (i.e., CARET, CARL, etc.) revealed architectural mismatches caused by the fact that each of the candidates were developed with a set of assumptions about the required architecture based on the anticipated use of the system and understanding of the developers. Each of the components was designed and developed as a closed system without providing any entry point except a User Interface, (UI). Apart from conflicting architectures, components may not fit easily together because of low-level problems of interoperability, i.e., different platforms, conflicting repository schemas, or incompatible programming languages [3]. The components to be integrated had these interoperability problems. Moreover, each of the components has its own proprietary process model and was aimed at providing its services as an independent system throughout the requirement management process. Making matters worse, there was hardly any documentation on architectural and implementation decisions taken for those systems.

Developing software systems by composing a number of independent components, each of which was designed and developed without any thought of reuse or potential integration, require a number of fundamental and complex design decisions regarding component interaction (e.g., communication, coordination) and their structural composition. What will be the respective role of each of the components in the resulting integrated software system? Which component will provide what service to other components and in which sequence services will be composed to provide an integrated service to the end-user? Will service composition be visible to the end-user or not? What integration techniques will be more appropriate to accomplish the required tasks cost-effectively and within available resources? Another important issue was to make a decision regarding the data repository of the integrated toolset. Each of the components had a repository with a very peculiar logical and physical structure. The two obvious options were to keep a repository for each component separate or develop a shared repository by reengineering the data models of the existing repositories.

Apart from these integration issues, there were a number of architectural, design, and implementation problems in the available components. Having analyzed the PARSNIP's requirements, existing components and available resources, it was obvious that it may not be possible to implement all of the components on a single platform using compatible technologies. We needed an architecture that would have interoperability and changeability embedded within it; this architectural perspective was driven by the requirement of developing a system that is easy to use, maintain, modify, and integrate with other tools. To answer all these fundamental questions and address other complex issues it was decided to focus on planning, designing, describing, evaluating and documenting an appropriate architecture. Having

decided to make the architecture of the new system as a cornerstone deliverable of our project, we planed, designed and analyzed an architecture that could ensure that the resulting system not only provides all the functions of independent systems in an integrated environment, but also meets a number of non-runtime requirements such as maintainability, enhanceability, and usability. As already mentioned we had to develop our tool by assembling independently developed components and that was why integrating those components in an effective and efficient manner was our primary concern. We attempted to design a new architecture that complies with the constraints imposed by those components; however, it also attempts to compensate some of the architectural weaknesses (such as inappropriate modularization, inflexible repository structure, interface and business logic codes intermingled and so on) found in the available components. This architecture based approach to build a software system by composing independent subsystems paid off in terms of rapid development and efficient use of project resources, which are some of the major benefits of architecture-based development reported in [12].

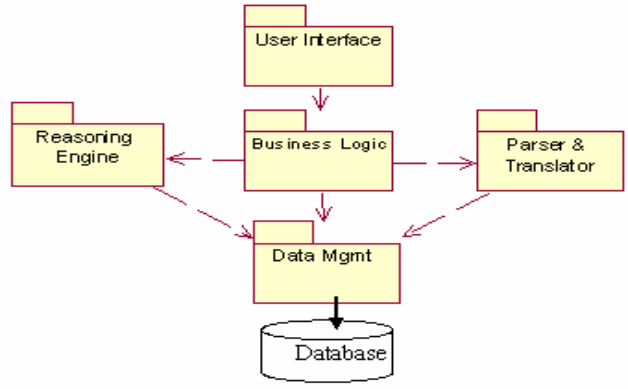

Figure 2 Logical Architecture of PARSNIP

4. The Software Architecture of PARSNIP

The most important goal of the project was to build a flexible and extendable toolset by integrating available independent tools. We decided to design an initial high-level architecture and keep refining it throughout the project lifecycle. We developed a logical model of the integrated toolset based on our understanding of the functionality required and knowledge of components and code to be used during implementation [18]. The high-level architecture of PARSNIP is shown in Figure 2. This logical architecture provides an abstract description of the gross structure of the system, its components, their organization and interaction with each other. The key components of this architecture are the presentation, computation and storage entities, i.e., GUI, integration components, natural language parser and translator, reasoning engine, share database, etc. These components interact with each other using standard connectors like client/server protocols, database queries and request/reply.

Though we are presenting only a logical view of our architectural description using package notation of Unified Modeling Language (UML)[22], we also developed other architectural views, e.g., development view, deployment view etc. Each view captures a specific set of concerns that are of interest to a given group of stakeholders [11]. The logical architecture provided the development team with a framework of reasoning about the capability of the architecture to satisfy quality requirements like maintainability, modifiability and reliability.

The architecture of PARSNIP is designed so that it not only allows a tight integration of the capabilities of individual systems, but it also supports extensibility and flexibility. During architectural archaeology of the individual systems, we classified their functionalities into two categories: unique services (consistency checking, requirement structuring, natural language processing, etc.) and generic functions (repository management, requirements management, etc.). Two of the systems being integrated also had their own models of requirements process and of attaching attributes to a requirement. The architecture of PARSNIP encapsulates the unique services in self-contained components. These self-contained components provide their respective functionality through well-defined interfaces whilst hiding implementation details [13]. We believe this modular approach will result in a highly maintainable and easily extensible system.

The interface layer* provides a uniform means of intering with the functions incorporated in the integrated environment, i.e., managing requirements, natural language processing, consistency checking, domain modeling, etc. This layer combines and enhances the functions provided by graphical and command-line interfaces of each of the systems that make up PARSNIP. This is one of the two components (the integration component is the other) of our tool that have almost been developed from scratch. The presentation component is responsible fro ensuring that the user's experience with the environment is as comfortable and uniform as possible. This component

* In this paper, we use Interface layer and presentation component as synonyms.

also provides a more refined model of the requirements process and the attaching of attributes to a requirement. There is a certain process model (described later) of managing requirements during various stages of the software development process using PARSNIP. The user is expected to adhere to this process model to fully utilize the toolset; this process control model is embedded in the interface layer and provides the user sufficient process control guidance. The reasoning engine component (previously CARET) maintains different sets of requirements and identifies a few semantically relevant operations that can be performed on requirements of a particular set [7]. This semantic model has been mapped onto the presentation layer in a more flexible manner than the previous implementation.

The integration component is a middleware layer that tightly integrates all components to form an integrated environment. This component is responsible for seamlessly gluing all the components together and for exposing the services of the shared repository, natural language parser, and reasoning engine components to a client component in a well-defined and controlled fashion. This client component can either be an interface layer as in PARSNIP or another system that requires the services of any of the components of PARSNIP. Apart from exposing the functionalities of other components, the integration component provides a number of housekeeping and data processing functions such as data validation, database connections, string parsing and manipulation, assigning unique identifiers etc. When a client requests a particular service, this component rigorously checks and validates the data and service request before forwarding them to the data repository or to an appropriate component based on the nature of the service required. It also receives the service failure or success message from the serving component, sends an appropriate database update request and informs the client of the result.

The natural language processing component processes the requirements and facilitates the process of building a domain model for a particular project. It accepts requirement sentences conforming to a particular grammar. It uses Cico to generate a parse tree for the sentence, and then translates that parse tree into logical notation. It traces the usage of phrases in requirement sentences, and prohibits the use of words marked as impermissible. It also generates an English paraphrase for the resulting logical form for user confirmation. In PARSNIP, this component provides its functionality in response to a message from the integration component. Based on the nature of the service request and the information arriving with the message, it retrieves the required data from the shared repository, processes it, stores the processed data back in the shared repository, and informs the caller of the results. We have developed our natural language component by customizing Cico, a natural language parser freely available to researchers, and by writing code to translate the parse trees that Cico generates into logical notation. Cico was developed primarily for use with domain-based grammars, however, we use a grammar based on English syntactic concepts instead. Details about Cico have been published in [15, 16].

The reasoning engine component is responsible for detecting any inconsistency in the requirements set. It also provides process support for managing requirements by associating requirements to different sets. Association with a particular set of requirements has a semantic value that is stored along with the requirement. The integration component reveals these semantics to the user through the interface layer; it is also used to mark the operations that are not permissible in a particular context. For example, if the reasoning engine has processed a requirement, general requirements management operations (e.g., editing or deleting) are not allowed unless the requirement is rolled back to pre-reasoning engine stage. When the integration component requires a service of the reasoning engine, it uses an appropriate interface to call the required service. The reasoning engine retrieves the required data from the repository component, performs the requested operations, stores the processed requirement in the repository, and informs the requester that the required operation has been performed. If the reasoning engine detects any inconsistency, it generates the maximal consistent subsets and asks the user to select one of the consistent subsets. An algorithm of checking and reporting any inconsistency between requirements of a requirement set has been based on proven and well-known theories of classical logic, non-monotonic reasoning and belief revision. The reasoning engine can only accept requirements as expressions of non-monotonic predicate logic; each logical expression must have a priority (a numerical value) attached to it and it may also have a description of the requirement in natural language (English only) as an attribute. Theoretical concepts used to develop the reasoning engine have been reported in [7, 17].

The shared repository component provides a centralized storage space to store the requirements, the domain model and the attributes attached to the requirements. It also provides centralized data manipulation and management services. Data related business rules have been implemented and stored on the shared repository to provide high performance,

security and consistency in data access operations. The shared repository component provides not only all of the storage and related functions provided by the individual components but also the functions required to enforce data manipulation logic and to keep track of the requirements when they are being processed by different components. Since a requirement is processed by different components throughout its existence in the repository, it is an easy means of sharing information between various components.

Instead of having a shared repository component, it would have been quite straightforward to allow each tool to retain its own original data repository, and transfer requirements and metadata between tools using a common data interchange format and an appropriate IPC mechanism. Such an approach could have saved a lot of effort required to reengineer the individual repositories to design an integrated repository. However, this solution would have required far more effort to write code to generate and parse the interchangeable files for each tool on each platform. Moreover, this solution did not seem easily extensible enough to accommodate any change in metadata or relationships among requirements. We have already mentioned that individual components had their respective repository systems, relational or ASCII files, with a very specific structure [17].

The shared repository has been designed to combine the storage mechanism of individual repositories of the components; and it emulates their peculiar structures to minimize the required modifications in each of the components. As a result of this structural emulation, there is some data duplication as the reasoning engine stores a requirement in logical notation while its natural language equivalent already exists in the repository. Keeping the data storage requirements of the reasoning engine separate from the other components of the tool required minimal code modifications, which resulted in a much cleaner solution and more rapid development.

5. Integration Techniques

Having designed and analyzed an architecture that can result in a system capable of meeting the functional requirements (Standard RM tool functions and inconsistency management) and non-functional requirements (maintainability and modifiability), we started evaluating various options of implementing the architecture. Again the logical architecture made a number of design decisions quite easy. As it is obvious from the logical architecture, we decided that our integrated environment would incorporate the GUI and requirement management functionalities of the tool X and consistency checking functionality of CARET; and the natural language parsing component would be used to help the user build a domain model and translate the requirements into a logical notation that is acceptable to the reasoning engine. That means the focus of our integration and development efforts was at three out of four levels of Enterprise Application Integration identified by D. Linthincum [5]: Data level, API level, and UI level. Having evaluated different approaches to make these components communicate with each other despite incompatible technologies and platforms, we narrowed our options to Data Integration and Control Integration techniques [14, 17].

Data Integration is a technique of developing shared data repositories to hold information that is shared by different systems [14]. This is a simple and well-established technique for integrating disparate organizational systems as long as all systems store, retrieve, and manipulate the information in a standard format. We needed to have a high level of integration among our tools to have them share their work to provide the user a coherent service. It was necessary to use a standard data format and structure. In this case, implementing a data integration technique was a very challenging issue as all the components to be integrated were using different database schemas and various types of software.

Control Integration is an approach to make different systems interoperate using a message passing technique. Tools integrated using this technique send messages to each other whenever they need to share some information or whenever a command is invoked from a tool that requires the services of another tool [14]. Message passing can be implemented either using a centralized server or point-to-point messaging. We decided to use a point-to-point message passing technique because of its simplicity and ease of development. We clearly defined the message protocols that tools would use to communicate with each other. Whenever a tool needs a service of another tool, it sends a request message along with a required service name and parameters to the tool, which can provide that service; in this scenario, the service seeker is called the client and service provider is called the server. When the server tool receives a service request, it processes the request and informs the client tool through a response message.

Our logical architecture and integration approach resulted in a flexible and scalable application whose components have a minimum amount of knowledge of the implementation details of each other. For example, the presentation layer has almost no knowledge about how data are validated, stored, and manipulated during

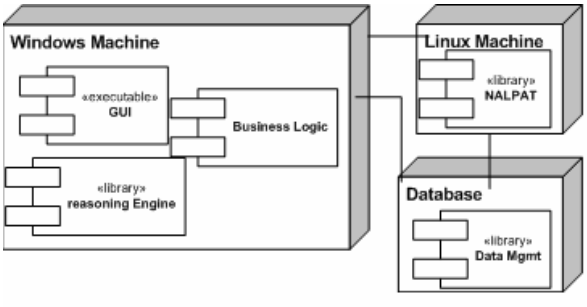

Figure 3 Deployment architecture of PARSNIP

the progression of requirements through various stages of their life cycle. Thus, it focuses on what its main responsibility is: providing the user with a uniform interface to perform a number of RM tasks. The integration component can easily be modified to access data sources of other requirement management tools that need natural language processing or consistency checking services.

The logical tiers of PARSNIP can be deployed on one or more physical tiers depending on the organizational requirements. A small organization with a modest number of projects, each with a few hundred requirements, can easily deploy PARSNIP on one or two machines. However, for large numbers of requirements, deploying the natural language parser and reasoning engine on dedicated machines will certainly result in reduced processing time. We have demonstrated that logical tiers of PARSNIP can successfully be deployed on one or more machines. Figure 3 shows one of the deployment options of logical tiers. In this deployment, PARNIP has been deployed on three machines: a Windows platform containing the presentation, integration, reasoning engine and shared repository components, a Unix cloned platform with the natural language parser and translator and another Windows platform hosting the database server and data management logic component.

6. Technological Choices

Given the time and resources available to the project team, we chose relatively inexpensive and less complicated techniques for integrating existing components or developing new ones. As mentioned, we felt that integrating all the components by using a shared data repository was a more appropriate technique to integrate the services of all the tools into an integrated environment. We re-engineered the individual data models of CARET and CASCAPS and develop a data model that not only serves the needs of the two tools, but also provides an integrated repository. A data model has been implemented using Microsoft Data Engine (MSDE) 2000. Since we decided to store data manipulation logic on the data service tier, MSDE seems quite natural progression from Microsoft Access. The natural language parsing component deployed on a Linux machine accesses the windows based data repository over the network. The database is used both to persistently store the data along with any metadata used by the application and to transfer the data and metadata between components deployed on the windows and Linux platforms.

Once we implemented a common repository for all tools, our next problem was to decide how to access the Windows-based database from a Linux platform. We considered two of the most reliable technologies being used for this purpose: Easysoft's Unix-ODBC Bridge and DBD::Proxy [24, 25]. We experimented with each, and decided on DBD::Proxy, since it was cost effective and appeared to do all that we required. Easysoft's Unix-ODBC Bridge may have advantages when dealing with a higher volume of transactions or more complex applications, but for PARSNIP, we decided to implement an open source solution rather than a commercial one as a number of research projects may not have the funds to buy commercial products. Additionally, we intended to port our Linux-based natual language component to Windows so that the whole toolset would be deployed and managed on a single platform.

The basic service, which the Linux-based component provides to the Windows-based component with, is the conversion of requirements sentences expressed in natural language (i.e. English) into the corresponding logic. Given a requirement ID, the Linux-based natural language component reads the requirement from the database, parses it, and then stores the requirement in logical notation back in the database. This communication is performed using the same table the Windows components use to store data persistently. Another issue was how to transmit the requirement ID to the Linux-based component and instruct it to retrieve the data from the shared repository and commence processing. We could have stored the parsing request and the requirement ID in the database, but how then would the Linux component know that it was there? It could poll the database at regular intervals-but regular polling would have resulted in a high number of accesses to the database and communications across the network, while infrequent polling would have made the application non-responsive to the user. We needed some kind of remote procedure call, so that the

Windows component could inform the Linux component that its services are required. We decided to use a Perl based technique called DBI as one of the team members had expertise in quickly implementing this technology.

DBI::Proxy uses a pair of Perl packages, RPC::PlServer and RPC::PlClient, part of the PlRPC distribution by Jochen Weidmann, to communicate between the Linux and Windows components. These packages implement a simple and Perl-specific remote procedure call protocol. Although it is more limited, has lower performance, and is less language-neutral than a more advanced RPC mechanism (such as ONC RPC, or CORBA) it is more than adequate for our application. Since we were already using RPC::PlServer and RPC::PlClient indirectly through DBD::Proxy, we might as well use it for this task also. The main program on the Linux side is called *parsnip-server.*This uses RPC::PlServer to listen for incoming network connections, and provides a procedure that can be called from the Windows machine to read a requirement sentence from the database, parse it and store the results back into the database. The client program, called *parsnip-client*, is located on the Windows machine (although it can be run under Linux also, where it was originally developed). Both of these programs have been written in Perl. The *parsnip-client* program takes a requirement ID as an argument and calls the parse procedure on the Linux machine running *parsnip-server*. It then saves to a temporary text file the status of the parsing processing (either "DONE" to indicate success, or an error message) returned by the remote procedure call.

When a client component, e.g. presentation layer needs to access the services provided by the natural language component, e.g. glossary generation, translation etc, it forwards its request through the integration component using a message communication technique to inform the parsnip-server. When the natural language component finishes processing the request it sends a message to the parsnip-client, which writes the appropriate message to a text file and finishes. The PARSNIP application reads the content of the text file and takes appropriate action based on the result of the parsing operation [23].

7. Lessons Learned

Requirements compromises may be inevitable - when building a system out of available components, all the stakeholders should be prepared for requirement compromises. The specifications of the system may need to be modified by taking into account the architectural capabilities and functionality of the components to be integrated. It may mean that some requirements compromises may have to be made. The functionality of the available components may not be a precise fit for the user requirements or there may be architectural tension between available components. In such situations, it is very important that user requirements are flexible and renegotiable.

An architecture-based approach to development pays substantial dividends - we decided to focus on the architecture of the new toolset and plan our integration activities around that architecture. Our experience with the architecture-based approach to manage a project was quite positive. Focusing on the architecture resulted in more refined and complete requirements for the integrated toolset. Discussions on the structure of the systems, its components and inter-component communication resulted in increased communication between stakeholders that caused better comprehension of the requirements. We designed our first cut architecture based on the domain model, knowledge of the functionality provided by each available system and analysis of the data models and application code. Then we successively refined our architecture as the interfaces of each component and the requirements for the integrated toolset became more comprehensible and clear. Our approach to focus on the architecture of the system resulted in a system that satisfies the required functional requirements as well as quality requirements like maintainability and modifiability. We demonstrated that the PARSNIP is easily modifiable by replacing the reason engine's original theorem-prover with another open source thermo-prover in just one man day effort.

Recognize the role of architectural documentation - a non-existence or lack of appropriate architectural documentation makes the application integration or system enhancement task very difficult and it may be the single greatest impediment to modifiability or maintainability of a system [29]. A clear and well written architectural documentation is considered one of the vital artifacts of a software project as it provides the basis for architectural reviews, implementation guidance, system evolution, and testing [31]. When we began analyzing the independent systems to be integrated, we found no documentation regarding the architectural or implementation rationale; a situation described as a common problem in [26]. We had to perform architectural archaeology by studying and analyzing the source code and functionality of the available components and by locating and discussing with the original architects/developers to find out the rationale for their design decisions. It was a painstaking process. Since we focused on the

architecture of the desired system, we documented our architectural decisions and their rationale using appropriate diagrams and models and kept this documentation in step with the system implementation decisions. The rigor in documenting the architecture resulted in improved architectural documentation that, we believe, will greatly facilitate the future modification or integration efforts.

Be ready to carry forward some of the wrong architectural decisions - when developing a system using available components or legacy systems, it is often easy and cost effective to carry forward some of the wrong design decisions made by the component developers. However, the effects of such decisions on the overall functionality of the integrated toolset must be carefully analyzed and documented along with the architectural documentation. Our experience has shown that it easy and less time consuming to workaround the rigid design decisions reflected in the source code of the components. Any attempt to correct the bad design may require a huge effort for code modifications and the gains may not be worth the effort. We saved a lot of time and effort by not attempting to correct the data model and class structure of the reasoning component. We decided to change only those parts of this component that required minimum code modifications.

Evaluate and choose open source and commercial technologies based on their appropriateness - We used open source and commercial technologies according to their capabilities to provide the functionality required for our integration project. For most of our implementations decisions, we evaluated both open source and commercially available alternatives and chose the one that promised to provide ease of modification, deployment and management within the time and monetary resources allocated to the project. For example, we decided to implement the tool's data repository on MSDE 2000 platform. We considered the original implementation on Access 97/2000, open source solutions like MySQL and Oracle9i as alternatives. However, we found MSDE 2000 more functionality-rich and scalable compared to open source solutions like MySQL and easy to deploy and manage compared to enterprise-level solutions like Oracle. Other reasons for using MSDE2000 were that it was a natural progression from MS Access97/2000, it was compatiable with operating system (Windows) and component infrastructure (COM+), and it can be freely distributed with applications built using Microsoft Technologies.

Learn from experiences and mistakes of others - Through literature reviews and peer discussions we found that there were a number of projects similar to our project that had been successfully completed. We realized that a number of aspects of a project could benefit from discussions with system integrators and architects of those projects. We gained very useful insights by discussing our design decisions and implementation alternative with them without compromising the confidentiality or commercial interests of our project. This lesson is especially important for projects undertaken in research and development institutes where budgetary and resource constraints are a norm of the workplace. Keeping close ties with the industry partners involved in the project and other peers who had already worked extensively with component technologies on integration projects proved quite beneficial.

Use prototyping to elicit and clarify requirements- It is quite possible that the customers or end-users may not be able to anticipate all the possible uses of the new system. Especially for large complex systems being developed either from scratch or through integration efforts, it is almost impossible to get clearly specified requirements upfront. In such situations prototyping is a useful technique to elicit and clarify the end-user requirements [21]. In particular, user interface prototyping is considered one of the most effective means of eliciting user requirements to improve the usability of an application [32]. At the beginning of our project, there were some uncertainties about the functions and features to be included in the integrated toolset; that was why we relied heavily on evolutionary prototyping, particularly in designing and implementing the user interface component and data model, and refining the parsing component. In this way, we not only accelerated the delivery of the application but also evaluated the side effects of any new function or feature to allow us accept or reject proposed changes in a cost effective manner.

8. Conclusions

This paper presents our experience of developing a component-based RM toolset, PARSNIP. We have shown how we successfully managed a number of issues caused by architectural mismatch, heterogeneous platforms, incompatible programming languages, and component specific repository structures by applying disciplined and proven architectural-centric integration approaches. The high level description of the SA not only provided a reasoning framework for detailed design and configuration of components but also guided the development process. The integrated toolset developed

using an architecture-centric approach is highly maintainable and easily modifiable.

9. Acknowledgements

This project was funded by an ARC SPIRT grant held by Prof. Offen and A/P Zowghi. The authors would also like to acknowledge the valuable insight provided by Vincenzo Gervasi and Simon Kissane on the natural language parser and translator component.

10. References

[1] A. Finkelstein and W. Emmerich. The future of Requirements Management Tools. An invited paper. Information Systems in Public Administration and Law. Austrian Computer Society, 2000.

[2] A. Wasserman. Toward a Discipline of Software Engineering. IEEE Software, November, 1996.

[3] D. Garlan, et al. Architectural Mismatch or, Why It's Hard to Build Systems Out of Existing Parts. Proceeding, 17th int. Conf. Software engineering, ICSE-17, April 1995.

[4] D. Garlan. Software Architecture: a Roadmap. The future of Software Engineering, A. Finkekstein (Ed), ACM Press, 2000.

[5] D. Linthicum, Enterprise Application Integration. Addison-Wesley, 2000.

[6] D. Zowghi, V. Gervasi, and A. McRae. Using Default Reasoning to Discover Inconsistencies in Natural Language Requirements. Proceedings of the8th Asia Pacific Software Engineering Conference (APSEC2001), Macau, China, December 2001.

[7] D. Zowghi. A logic-based framework for the management of changing software requirements, PhD Thesis, Macquarie University, Sydney, Australia, 1999.

[8] L. Bass, P. Clement, and R. Kazman. Software Architecture in Practice. Addison Wesley, Reading, Massachusetts, 1998.

[9] L. Bass, R. Kazman. Architecture-Based Development. Technical Report, CMU/SEI-99-TR-007, ESC-TR-99-007. SEI, Carnegie Mellon University, April 1999.

[10] N. Medvidovic, D. Rosenblum, and R. Taylor. An Architecture-Based Approach to Software Evolution, IWPSE, 98, Kyoto, Japan, April 1999.

[11] P. Clements, L. Northorp. Software Architecture: An Executive Overview. Technical Report, CMU/SEI-96-TR-003. SEI, Carnegie Mellon University, 1996.

[12] P. Clements. Coming Attractions in Software Architecture. Technical Report, CMU/SEI-96-TR-008. SEI, Carnegie Mellon University, 1996.

[13] R. N. Taylor, et al., Foundations for the Arcadia Environment Architecture. 3rd ACM SIGSOFT/SIGPLAN Symp. Practical Software Development Environments. 1988.

[14] S. Reiss. Software Tools and Environment, ACM Communication Surveys. Vol.28, No. 1, March 1996

[15] V. Ambriola and V. Gervasi. Experiences with domain-based parsing of natural language requirements. In G. Fliedl and H. C. Mayr, editors, Proceedings of the Fourth international Conference on Applications of Natural Language to Information Systems, number 129 in OCG Schriftenreihe (Lecture Notes), 1999.

[16] V. Gervasi. Environment Support for Requirements Writing and Analysis. PhD thesis, University of Pisa, 2000.

[17] V. Gervasi. On the integration of CASAPS and CARET. Personal Communication, June, 2001.

[18] P. Clements et. al. Constructing Superior Software. ed. S.Q.I. Series. 2000, Macmillan Technical Publishing, U.S.A.

[19] B. Boeham, and P. Grunbacher, and R. O. Briggs, Developing Groupware for Requirements Negotiation: Lesson Learned, IEEE Software, May/June 2001 18(3)

[20] B. Nuseibeh and S. Easterbrook. Requirements Engineering: A Roadmap. in The Future of Software Engineering, 22nd International Software Engineering Conference. 2000: ACM-IEEE.

[21] I. Sommerville. *Software Engineering*. 6th ed. 2001: Addison-Wesley.

[22] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*. 1999: Addison-Wesley.

[23] Microsoft Knowledge Base article # 191, Shelling to Other Applications http://msdn.microsoft.com, accessed 20th June, 2001

[24] Perl DBI/DBD FAQ on the Web http://tlowery.hypermart.net/perl_dbi_dbd_faq.html, accessed on 20th August, 2001

[25] Easysoft Unix-ODBC Bridge Documentation, http://www.easysoft.com, accessed 20th August, 2001.

[26] R. Kazman et al. Experience with Performing Architecture Tradoff Analysis. in Proceedings of the 21th International Conference on Software Engineering. 1999. New York, USA: ACM Press.

[27] K. E. Wiegers. Automating Requirements Management, Software Development, July 1999, http://www.sdbestpractices.com accessed 28th Sep., 2002.

[28] A. Fuggetta. A Classification of CASE Technology, IEEE Computer, 1993. 26(12).

[29] N. Lassing, D. Rijsenbrij, and H. V. Vliet., The goal of Software Architecture Analysis: Confidence Building or Risk Assessment, in Proceedings of the First BeNeLux Conferencde on Software Architecture, 1999.

[30] B. Gautier, C. Loftus, and E. Sherratt, Tool Integration: Experiences and Directions, in Proceedings of the 17th International Conference on Software Engineering. Apr., 1995, Seattle, Washington, U.S.A., ACM Press.

[31] D. Garlan, and J. P. Sousa. Documenting Software Architectures: Recommendations for Industrial Practice. Technical Report, CMU-CS-00-169 School of Computer Science, Carnegie Mellon University, Oct., 2000.

[32] Wasserman, A. I., Toward a Discipline of Software Engineering, IEEE Software, 1996. 13(6).