Centrum voor Wiskunde en Informatica

Collaborative software development

M. de Jonge, E. Visser, J.M.W. Visser

Software Engineering (SEN)

SEN-R0113 May 31, 2001

المنارة للاستشارات

www.manaraa.com

Report SEN-R0113
ISSN 1386-369X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

CWI is the National Research Institute for Mathematics and Computer Science. CWI is part of the Stichting Mathematisch Centrum (SMC), the Dutch foundation for promotion of mathematics and computer science and their applications.
SMC is sponsored by the Netherlands Organization for Scientific Research (NWO). CWI is a member of ERCIM, the European Research Consortium for Informatics and Mathematics.

Copyright © Stichting Mathematisch Centrum
P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

المنارة للاستشارات
www.manaraa.com

Collaborative Software Development

Merijn de Jonge
email: Merijn.de.Jonge@cwi.nl
CWI
P.O. Box 49079, 1090 GB Amsterdam, The Netherlands

Eelco Visser
email: visser@acm.org
Universiteit Utrecht
Institute of Information and Computer Science
P.O. Box 80089, 3508 TB Utrecht, The Netherlands

Joost Visser
email: Joost.Visser@cwi.nl
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT
We present an approach to collaborative software development where obtaining components and contributing components across organizational boundaries are explicit phases in the development process. A lightweight generative infrastructure supports this approach with an online package base, and several generators that simplify the construction and composition of component packages. The infrastructure ensures availability, portability, and adaptability of components without centralized orchestration of the development process.

1998 ACM Computing Classification System: D.1.1, D.2.1, D.2.2, D.2.6, D.2.7, D.2.9, D.2.13, D.3.3
Keywords and Phrases: Collaborative, software development, software bundling, component based

1. Introduction

Reusability of components lies at the heart of component-based software development. The reusability of a component is determined in part by the functionality it implements. If the functionality is more generic, its potential for reuse is higher. But even for highly generic components, the actual scope of reuse can be severely limited. In fact, the reusability of components is largely determined by issues unrelated to their functionality.

- *Availability*: If I know or suspect certain functionality has been encapsulated in a component, will I be able to find and obtain this component?
- *Portability*: Can I lift a component from its initial context and insert it into mine? Or does the component silently assume a specific platform, specific peer components, or specific development, compilation, run-time, test, or documentation environments different from mine?
- *Adaptability*: If the component's functionality is close to what I need, but slightly off, can I adapt it to my specific needs? When developing adaptations, can I reuse the original tests and programming aids as well as the source?
- *Contribution*: If I develop components myself, can I make them reusable by others? How much effort will that take from me, and from them?

المنارة للاستشارات
www.manaraa.com

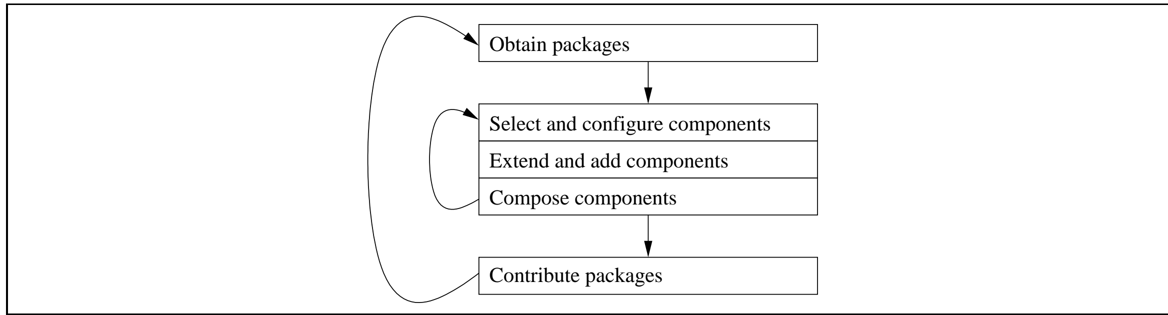

Figure 1: Collaborative component-based software development can be described with a development cycle with explicit phases for obtaining packages from, and contributing them to a shared repository.

These practical issues demonstrate that component reuse involves much more than multiple invocations of the same code. The issues of *adaptability* and *contribution*, in particular, venture beyond 'as-is' reuse into *collaborative use and development*.

To address non-functional as well as functional issues surrounding component reuse, we have developed a range of generative techniques for collaborative development of software components. These techniques enable easy component sharing across project and organizational boundaries. Section 2 gives an overview of our approach, including terminology and a schematic development cycle. Sections 3 to 6 explain the generative techniques underlying each step in the development cycle.

Our techniques were developed in the course of about 18 months in the context of the development of the transformation tool bundle XT [5]. The initial purpose of XT was to (i) distill a set of components from previous language tool development projects and make their reuse as easy as possible, and (ii) to demonstrate this reusability by developing a suite of program transformation tools on top of these components. As our techniques are not specific to the program transformation domain, we describe them independently from XT. Nonetheless, XT remains a prime example, implementation medium, and breeding ground for further collaborative development techniques.

2. A Collaborative Development Process

Terminology Software components can be defined as 'building blocks from which different software systems can be composed' [3]. Examples of components at run-time and compile-time are executables, source modules, and libraries. In the context of collaborative component development, it is useful to additionally distinguish a notion of component at distribution-time. Throughout the paper we will use the word *package* for this notion. A package, then, is a source distribution of a set of (compile-time) components, together with tools that automate their compilation, testing, and distribution processes. These tools constitute the *build environment* of the components.

Development Cycle Figure 1 gives a schematic overview of the collaborative component development process. In this figure package exchange is mediated by a repository of reusable software packages, called a *package base*. In the first phase, those packages that contain components of interest are downloaded from the package base, compiled and installed. Secondly, the components of interest are identified within the installed packages, and they are configured for specific purposes. Third, new functionality is implemented by extending some of the components, or developing additional ones. Fourth, a specific application is implemented by capturing the component configuration and composition knowledge in an integration script, which may itself be viewed as a higher-level component. Finally, changes to downloaded components and additional components are made available for reuse by others, by contributing them to the package base. Steps two, three and four are performed in frequent, short iterations (inner cycle), and their boundaries may be unsharp. The two outer steps of uploading and downloading packages are performed in long, infrequent iterations (outer cycle).

Note that the possibility of extending components obtained from other sources requires that these

www.manaraa.com

```
package
identification
  name     = cpl
  version  = 0.1
  location = http://www.loria.fr/equipes/protheo/SOFTWARES/ELAN/Soft/
  description =
      'Choice Point Library. Functions for efficient choice point
       management to support backtrack programming.'
  keywords = choice point, nondeterminism, backtracking
requires
    aterm 1.4.5
```

Figure 2: An example of a package description file for the `cpl` package.

components are available as open source (at least within the desired reuse scope), and that code ownership is absent. The infrastructure needed to participate in component exchange should be minimal, and should be as easily available as the components themselves. Preferably, (parts of) the infrastructure should be componentized for easy reuse.

3. Contribution of Packages

We start with general considerations about sensible selection of reusable components. Then we discuss our own situation, where a number of components were distilled from existing systems. We describe the steps that need to be taken to contribute such components to an online package base.

Start from Legacy In general, the development does not start in a vacuum. Various useful systems may have been built and are probably still being maintained in different projects. Within these systems distinct components may even be distinguishable at binary or source code level. Most likely, these component have even been reused in more or less ad-hoc fashion across system or project boundaries. Therefore, the best way to initialize systematic development of reusable components is to isolate these 'old' components, adapt them to the reuse infrastructure, and make them easily and more widely reusable by contributing them to the package base.

From Legacy to Reusable Component Once a candidate component or a few closely related components have been identified in an existing system, they are formed into a software package by the following steps.

1. *Automate configuration, compilation, testing and distribution:* Our infrastructure requires that the generators automake [10] and autoconf [9] are used for high-level definition and configuration of automated compilation, test, and distribution processes.

2. *Isolate source:* Often, components are clearly distinguishable at run-time while their sources are spread out through the source tree of the entire system. These sources need to be disentangled and isolated into separate source trees.

3. *Create a distribution:* Once an isolated source tree is available whose configuration and compilation have been automated with automake and autoconf, a source distribution can be created with a single command.

4. *Form a package:* To form a package, the source distribution must be made available for download, and a description of the package, including its download location, must be uploaded to the package base. An example of a package description file is given in Figure 2.

Note that only package *descriptions* are uploaded to and stored in the package base. In Section 4, we will explain how the information in these descriptions is used to generate self-contained bundles of packages.

www.manaraa.com

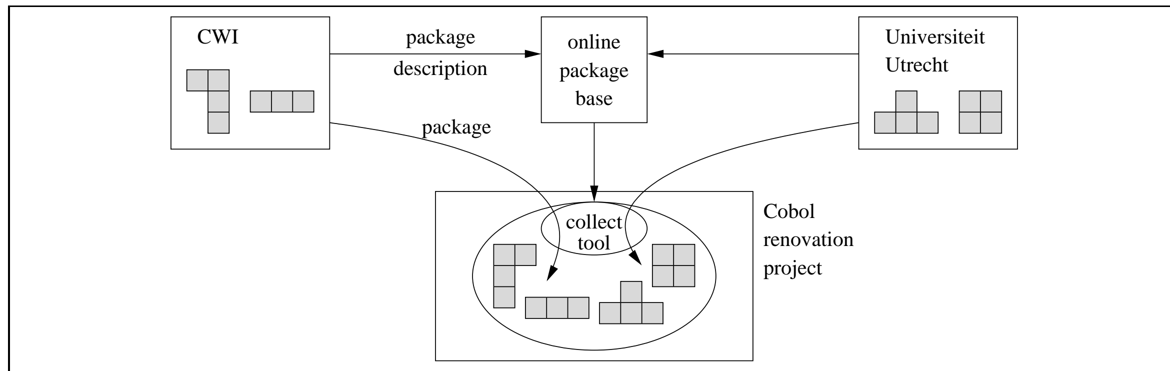

Figure 3: Obtaining package bundles from the online package base.

Contribution of Components In our own software engineering research groups at Universiteit Utrecht and CWI, a number of systems had been developed over the years in the context of numerous projects related to the groups' foci: program transformation and optimization, and interactive software development and renovation. We observed that many of the tools developed in these projects were built in a more or less component-based fashion, but nonetheless reuse across project boundaries proved extremely problematic. As a result of these problems, components were often not reused, or their reuse was performed in a cumbersome, ad-hoc fashion, involving code duplication and scavenging.

According to the procedure sketched above, we have been able to gather a diverse range of packages. These include, for instance, parsing and pretty-print components, the compiler and standard libraries of the rewriting language Stratego, a collection of modular grammars, the distributed process coordination architecture ToolBus, and more. By contribution of these packages to the package base, many of the core assets of our groups have become easily reusable across project and even organizational boundaries.

4. Obtaining Packages

In this section we will explain how a bundle of packages can be obtained through an (online) package base. Generative techniques are used to integrate the compilation, testing, and distribution processes of the different packages in a bundle. This phase of the development cycle is illustrated in Figure 3.

Shopping for Packages In order to obtain a specific bundle of packages from the online package base, one goes through the following steps:

1. Select the packages from an HTML form, and press the 'bundle' button.
2. The browser pops up a window for saving the generated bundle to your file system. The bundle still does not contain the selected packages, but only a tool to collect them.
3. Unpack the file, to obtain an 'empty' bundle, including the collection tool.
4. Invoke the collection tool to fully automatically download, unpack, and integrate all selected packages into a single build environment.

For instance, the screenshot of Figure 4 shows the selection of the ToolBus and Stratego packages.

Package Form The HTML form, which the user fills out in step (1), is generated automatically from the available package description files. The generator takes care of grouping descriptions of different versions of the same package. When a new description is uploaded, the form is updated by regeneration.

المنارة للاستشارات
www.manaraa.com

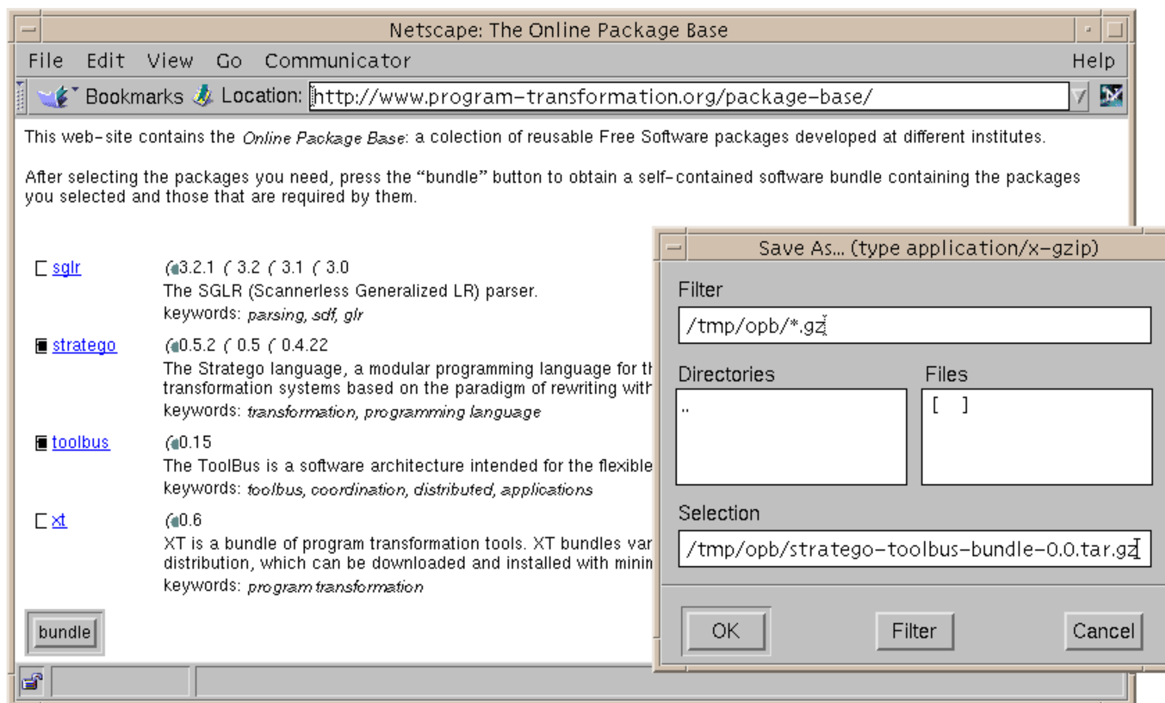

Figure 4: Filling out the package selection form of the online package base available at `http://www.program-transformation.org/package-base/`.

Bundle Generation When the user presses 'bundle', a bundle is generated from the form and the package descriptions in the package base. The generator takes care of the following:

- *Resolution of package dependencies*: Apart from the packages selected by the user, all packages they depend on, as declared in the `requires` clause of their description, are transitively selected as well. The declared version requirements are taken into account. The resulting list of packages is used to configure a generic collection tool.
- *Build environment integration*: After collecting the packages, their separate build environments must be integrated into a single build environment for the entire bundle. This is supported by generation of a bundle-specific configuration file, with which a generic build environment is instantiated. The generated configuration file performs partial configuration of a package when some other package requires so according to its description. Also, similar switches across packages are unified in bundle switches.

Note that HTML forms and bundles are generated by the package base server, so this software is not needed to participate in the package exchange via the package base. Nonetheless, this software is available as a package itself, under the name autobundle.

On the client side no additional software is required. The installation procedure of a software bundle consists of two user actions: specifying an installation location, and starting the build process.

5. Local Component Development

The inner cycle of the component development process is influenced by the fact that it takes place within a larger collaborative process. In particular, an integrated build environment is obtained together with the packages in a bundle. This implies that the support for configuration, building, testing, and distributing components is reused as well as the components themselves. In this section

المنارة للاستشارات
www.manaraa.com

```
# Give a brief description of your grammar.
description = Grammar for the eXtensible Markup Language (XML)
# Choose a maturity level (Volatile, Stable, or Immutable).
maturity    = Volatile
# Specify the extension of term files for your grammar.
suffix      = xml
testset     = preamble.xml suspect.xml remarks.xml GraphXML.dtd
```

Figure 5: A filled out grammar configuration form is used by the gbadd tool to automatically add a new grammar to the Grammar Base.

we will explain various generative techniques that we developed for configuration, building, and testing. Distribution techniques will be explained in Section 6.

Configuration For software configuration we depend on autoconf, an open source software package for the generation of configure scripts. Configure scripts perform system checks to verify that a platform fulfills all requirements for building a package. Furthermore, they offer switches to activate or configure parts of a package.

Though autoconf allows configuration at a high level of abstraction, its input is still on the level of directories and files. To abstract over these, we developed additional generators that introduce a notion of *component*.

- *Package configuration*: We introduced *package configuration files* that define the contents of a package by listing its components and their status (either as released or as unreleased component). From these files, we generate autoconf input files, and the components are added to the generic build environment.
- *Component configuration*: We introduced *component property files* in which component specific parameters are declared, such as its name, version, test set, and source modules. From these files, make rules and test scripts are generated for the component.
- *Configuration initialization*: We developed an interactive tool that generates the component's initial source subtree, including its component property file, and adapts the package configuration file, whenever a new component needs to be added.

For example, addition of grammars to the Grammar Base package is supported by the tool gbadd. It takes a grammar name and version, and offers the form as depicted in Figure 5 to the user. After the form has been filled out, the build environment for the grammar is set up (required directories are created and automake Makefiles are generated) and integrated into the build environment of the Grammar Base.

Building As mentioned before, we rely on automake to automate the software build process, including compilation, testing, and distribution. By generating complete Makefiles from concise automake input files, automake offers several abstractions over the build process, e.g. how to obtain an executable from a set of source files, and how to build distributions.

We observed that maintainability of automake files (like ordinary Makefiles) rapidly decreases as projects get larger. This is caused by code duplication, increasing complexity, and a decrease in readability. To improve this situation, we developed a few highly reusable automake modules, and standardized their use. As a consequence, we reduced each component-specific automake file to a concise, purely declarative Makefile module.

For example, in the Grammar Base package, the build process for grammars is expressed in a single automake module, which is reused for every grammar. Changing the build process of the package only requires editing this file. Per grammar, a single Makefile contains the declaration of the grammar parameters.

المنارة للاستشارات
www.manaraa.com

Testing The absence of centralized orchestration of the collaborative development process implies that changes to components should preserve backward compatibility. To support this, extensive automated regression tests should be shipped with components. Automake offers a mechanism for declaring individual tests and running them collectively. The individual test scripts themselves must be supplied by the developer.

To simplify testing, we developed generators that produce test scripts and test data required by automake's test mechanism. Test script generators consume a declaration of input to test, and optionally reference output. Test data generators produce regression data as a snapshot of a component's current functionality. Due to these generators, the effort to initiate and extend component tests has been significantly reduced.

Development The generative techniques for configuration, building, and testing of components are part of the integrated build environment that is obtained as part of a bundle of packages. As such, these techniques form the backdrop for the inner phases of collaborative development: selection and configuration of components, addition and extension, and composition of components. For instance, when a component is extended with new functionality, the build environment supports running the regression tests of the component, and updating them to test the new functionality. When new components are added, generators support their insertion into the build process with reuse of generic build support. Composition of components from different packages is facilitated by the standardization and integration of the build environments of these packages.

6. Contribution of Packages (Revisited)

When, after a number of iterations in the inner cycle, the development converges on a set of stable components, one may decide to make them more widely reusable, and re-enter the initial phase of contributing components to the package base. Contributions can be mutations to existing components, addition of new components to existing packages, or addition of new packages. The integrated build environment that comes with a bundle of packages contains various kinds of support for performing such contributions.

Selective Distribution Automake supports rules to automatically build a distribution from the information contained in automake files. However, automake's support for distribution generation is not very flexible: every component defined in an automake file will be distributed. This prohibits the use of automake for components that are still under development and not ready for distribution. To preserve stability of a package distributions on the one hand and to benefit from the build environment for the development of new (unstable) components on the other hand, we added support for selective distribution via the package configuration files described in Section 5. In this file, each component can be declared to be distributable or not.

Per-package or Self-contained Distribution After a (selective) distribution has been generated, it can be contributed to the package base by uploading a package description and by making the distribution itself available at a public download location. Such per-package distribution and contribution to the package base allows further development of the package by the same or other development groups, via new iterations of the development cycle.

As an alternative to per-package distribution generation, generation of self-contained distributions is also supported by the build environments of software bundles. Self-contained distributions are generated in order to ease download and installation of packages for use, rather than for development. From these self-contained source distributions, binary distributions can be generated also. Currently, Sun's package format and the RPM format are supported.

7. Concluding Remarks

Contributions We presented an approach to collaborative software development where obtaining components and contributing components across organizational boundaries are explicit phases in the development process. We discussed a lightweight infrastructure to support this approach. The main

المنارة للاستشارات
www.manaraa.com

ingredients in the implementation of this infrastructure are an online package base, and several generators that simplify the construction and composition of component packages. The infrastructure ensures availability, portability, and adaptability of components without centralized orchestration of the development process.

Experiences We initiated a package base to which, in the course of 4 months, 21 packages were contributed by 3 different organizations. On the basis of these packages, a number of applications have been developed, among which (i) a product line for Cobol legacy system renovation [12], (ii) a documentation generator for a proprietary dialect of the specification and description language SDL in collaboration with Lucent Technologies [4], and (iii) an implementation of a transformation system for a subset of the Haskell functional programming language [8].

Related Work Many open source projects involve some level of collaborative development. Examples are the Apache [1] and Mozilla [11] projects, the development of the GNU tools [6] and the Linux kernel. The distribution infrastructures employed by these projects are insufficient for collaboration across project boundaries. For instance, automake [10] and autoconf [9] automate and standardize the configuration, build, test, and distribution processes for a *single* software package, but we needed to supplement them with additional generators to support package composition with *package dependency resolution* and *build environment integration*. Dependency resolution is supported by package managers, such as RPM [2], but these are primarily intended to simplify distribution and installation of *binary* packages. They offer no (or only very limited) support for the integration of the configuration, build, test, and distribution processes of *source* distributions, which is essential for collaborative development. In contrast, our infrastructure supports modification and extension of downloaded packages, and redistribution either per package, or as self-contained, possibly binary, distributions. The Open Software Description format (OSD) [7] is a language for describing software components and is similar to the package description format used to describe the packages in the package base. However, OSD is designed for describing binary packages and their dependencies in order to simplify software installation over Internet, not for describing source packages to allow composition of source trees as needed for collaborative development.

Future Work Currently, component selection is done in two steps: first packages are obtained from the package base, and then components of interest are selected from these packages. We intend to implement a more sophisticated component selection mechanism where components can be selected directly, while their retrieval from specific packages remains hidden. Moreover, we want to support component selection by more advanced search and navigation. These improvements would bring component selection to a yet more generative (i.e. problem-oriented rather than solution-oriented) level.

المنارة للاستشارات
www.manaraa.com

References

1. Apache web server. Apache Software Foundation. http://www.apache.org.
2. E. Bailey. *Maximum RPM.* Red Hat Software, Inc., 1997.
3. K. Czarnecki and U. W. Eisenecker. *Generative Programming – Methods, Tools, and Applications.* Addison-Wesley, 2000.
4. M. de Jonge and R. Monajemi. Cost-effective maintenance tools for proprietary languages. submitted for publication, jan 2001.
5. M. de Jonge, E. Visser, and J. Visser. XT: A bundle of program transformation tools. In M. G. J. van den Brand, M. Mernik, and D. Parigot, editors, *Workshop on Language Descriptions, Tools and Applications (LDTA'01)*, Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers, April 2001. To appear.
6. The GNU project. http://www.gnu.org.
7. A. v. Hoff, H. Partovi, and T. Thai. The open software description format (OSD), 1997. http://www.w3.org/TR/NOTE-OSD.html.
8. P. Johann and E. Visser. Warm fusion in Stratego: A case study in the generation of program transformation systems. *Annals of Mathematics and Artificial Intelligence*, 29(1–4):1–34, 2000.
9. D. Mackenzie, R. McGrath, and N. Friedman. Autoconf: Generating automatic configuration scripts, 1994. http://www.gnu.org/autoconf.
10. D. Mackenzie and T. Tromey. Automake. http://www.gnu.org.
11. Mozilla web browser project. The Mozilla Organization. http://www.mozilla.org.
12. H. Westra. CobolX: Transformations for improving Cobol. In *Proceedings of the Second Stratego Users Day*. Technical Report, Univeriteit Utrecht. To appear, 2001.

المنارة للاستشارات

www.manaraa.com