An Internet-Scalable Knowledge Base Infrastructure

Kevin Jameson

Last Updated October 1997

Abstract

This paper describes an Internet-scalable knowledge base infrastructure for managing the knowledge used by an intelligent software productivity infrastructure system. The infrastructure provides workable solutions for several significant issues: (1) Internet-unique names for pieces of knowledge; (2) multi-platform, multi-language support; (3) distributed knowledge base synchronization mechanisms; (4) support for extensive customized variations in knowledge content, and (5) knowledge caching mechanisms for improved system performance. The infrastructure described here is a workable example of the kind of infrastructure that will be required to manage the evolution and reuse of millions of pieces of knowledge in the future. **Keywords**: Internet knowledge base, Internet knowledge reuse, software reuse.

Contents

1 Introduction

This paper addresses the overall problem of managing knowledge bases in an Internet-scalable way. It discusses issues of knowledge creation, representation, distribution, evolution, access, security, and replication—all distributed—in an Internet environment.

Information presented here is based on our experience in constructing an Internet-scalable infrastructure of intelligent software engineering tools. The infrastructure contains models, tools, and processes that provide for the creation, evolution, testing, distribution, and automated reuse of large numbers of software components.

This paper is limited to the knowledge management subsystem of the infrastructure. Other aspects of the *Software Productivity Infrastructure* `<http://www.realcase.com>` are described elsewhere.

1.1 Problem Description

Our research focus is to build intelligent, automated, Internet-scalable software engineering systems. In particular, our current goal is to build *An Internet Scalable Software Reuse Infrastructure* `<http://www.realcase.com/papers/ireuse.html>` capable of managing millions of software components.

We use an *infrastructure* approach to solving the software productivity problem, rather than the traditional *divide-and-conquer* approach. An infrastructure approach uses a shared knowledge base and integrated systems of coordinated programs to address the problem of cradle-to-grave software productivity.

In contrast, a traditional *divide-and-conquer* approach tends to use standalone tools to address selected (and usually narrow) parts of the software lifecycle. A compiler is an example of a divide-and-conquer standalone tool. An operating system is an example of an infrastructure approach.

The rest of this paper describes the shared knowledge base.

1.2 Infrastructure Look And Feel

This section summarizes some interesting capabilities of the infrastructure that are supported by the knowledge base. Look and feel scenarios are used to describe the capabilities in a practical, non-conceptual way. The next section summarizes the knowledge base features and operations at a conceptual level.

Suppose a developer wants to create and install a new client-server application on N platforms. Using infrastructure program commands, the developer would:

1. Create a container pnode to hold the new application.
2. Instantiate a multi-platform template application inside the container.
3. Check the working templates into a distributed version control system.
4. (Auto) Generate a custom, context-sensitive makefile for the situation.
5. (Auto) Build and install the application locally.
6. (Auto) Propagate code to remote sites using synchronized repositories.
7. (Auto) Build and install the application on remote sites.
8. (Auto) Extract software API and internal documentation.
9. (Auto) Propagate extracted documentation to remote sites.

One example of knowledge use in this scenario is the generation of custom makefiles. The makefile generation tools (1) analyze the current generation situation (the context), (2) analyze the contents of the pnode container (i.e. the contents of the application template), (3) analyze the developer/team/site preferences in the knowledge base, and then (4) calculate and instantiate a custom makefile based on all this knowledge. Once the context has been determined, all analyses and knowledge lookups are done in a context-sensitive way.

Contexts are purpose-oriented mechanisms that allow the knowledge base to return different answers to the same question, depending on the context. For example, suppose developers declared a "debug" context to be in effect at makefile generation time. Then knowledge base lookup operations could return sets of compiler and linker arguments that would cause symbolic debugging information to be retained in generated binary files.

The final makefile knowledge content is based on information from the context, from the pnode meta-data, from the pnode contents, from the developer/team/site/default infrastructure preferences, from the syntaxes required by all tools referenced by the makefile (including the native **make** tool itself, and from the local filesystem locations that participate in the processes described in the makefile.

A second example of knowledge use in this scenario is the automatic propagation, regeneration, and installation of software applications at remote sites. Makefiles are intelligently generated at all remote sites, using the (usually different) developer/team/site preferences, compilers, locations, and administrative policies that are active at each of the remote sites. In addition to makefile generation knowledge, much other knowledge is used to represent site preferences for host names, platforms, filesystem locations, and security policies.

This scenario happened to use the familiar example of creating a computer program (as software files) and building/installing it (with processes). But in general, this scenario applies to any arbitrary collection of software files and associated processes that manipulate them.

2 Knowledge Base Design Goals

The main goal of our research is to build an Internet-scalable software engineering infrastructure that can manage millions of software components. A credible knowledge base for this problem domain, in our view, requires support for at least the following things:

New Knowledge Creation. Models and mechanisms for creating new pieces of knowledge from distributed, reusable knowledge templates and fragments. A solution without the ability to use remote knowledge to create new knowledge is limited in network space, because local knowledge generation activities cannot use remote knowledge.

Models For Both Data and Process Knowledge.. Models and mechanisms for both passive knowledge (data) and active (process) knowledge. A solution that can only model passive knowledge is limited in knowledge space, because it cannot represent processes, nor execute dynamically constructed processes to obtain interesting results.

Extraction of Knowledge Documentation. Models and mechanisms for generating, distributing, and accessing documentation for large numbers of knowledge instances spread across the Internet. A solution without this capability provides an undocumented—and therefore ultimately intractable—user interface to stored knowledge.

Distributed Storage and Access. Models and mechanisms for storing, distributing, and accessing large numbers of knowledge fragments across the Internet. A solution without these capabilities is a non-scalable, local solution only.

Distributed Evolution. Models and mechanisms for managing the evolution of software components that are geographically distributed across the Internet. A solution without these capabilities is limited in both time (no past revision histories) and network space (only local solutions can be managed).

Appropriate Security. Appropriate security controls such as authentication and authorization mechanisms must be built into all system access and transport functions.

Appropriate, Convenient Interfaces. Models and mechanisms for browsing and retrieving knowledge contained in the distributed knowledge base. A multi-platform software API for accessing knowledge from client programs is mandatory for effective use of the knowledge by programs. A command line interface is mandatory to support use of the knowledge base by scripts, by servers that run external programs to retrieve knowledge, and by humans interested in browsing the knowledge base. A GUI interface is convenient for interactive human exploration of the knowledge base.

Optimized Knowledge Lookup Performance. Models and mechanisms for improving lookup performance for both local and remote knowledge lookups. Local caching of remote knowledge lookups and their results is a major performance mechanism. A solution without reasonable performance optimization mechanisms can be excruciatingly slow when performing knowledge lookups on remote machines across the Internet.

3 Drivers Of Knowledge Base Complexity

Our experience over the past five years of building and using this kind of knowledge base has shown that the three most important drivers of knowledge base complexity are scale, diversity, and customization.

Scale means "more of the same", or "more of a similar kind of thing." Scale is a quantitative effect. Increases in scale can often be treated by a corresponding increase in the capacity of existing mechanisms, such as more computing horsepower, more subdirectories in the knowledge base, more access keys in the knowledge base, more network bandwidth, and so on.

Diversity means "different stuff", or "more of a different kind of thing." Diversity is a qualitative effect. Increases in diversity cannot usually be treated with existing processing mechanisms. Instead, diversity usually requires the extension or modification of existing mechanisms to accommodate the new diversity.

Customization Diversity is a qualitative effect. It means that everything must have a default value, and one or more methods for customizing that value in a permanent, semi-permanent, context-driven, and per-invocation manner. Diversity in override methods is also important.

4 Implementation Lessons

Our experience has reinforced the following conclusions for the problem and knowledge domain that we work with:

4.1 Problem Domain and Knowledge Diversity

- Customization of knowledge content is mandatory for our diverse problem domain. In our experience, solving the large scale software reuse problem is impossible without a knowledge base that supports *extensive* customization of all knowledge involved.

- Even the access mechanisms to the knowledge base itself must be customizable. This allows users of the knowledge base some measure of control over how they will fetch knowledge. In our particular case, users can access the knowledge base through local NFS file read operations, through interactions with remote knowledge servers, interactions with local knowledge caching servers, or through some blend of the three.

- Multi-part lookup keys are required to handle the diversity of knowledge in this problem domain. Our knowledge base required a minimum of 5 levels of specialization in order to handle the diversity and complexity inherent in the domain problem. Each lookup request proceeds through these five levels of specification: search rules, context, virtual platform, program, and file/key pair. A sixth level of specialization is required for the makefile generation problem.

- Furthermore, sequences or chains of multi-part keys lookups are often required to fulfill a knowledge lookup. This is because retrieved knowledge must sometimes interact with dynamic problem information in order to specify further lookup steps to find the desired information. Chained lookup operations forced us to build special representations and implementations for acceptable lookup performance on remote servers. We now represent chained lookups as atomic operations, to avoid multiple round trip costs of individual lookup operations performed against remote servers.

4.2 KB Development and Deployment

- Distributed and automated tool support is mandatory for a knowledge base of this size. In particular, automated mechanisms are required to manage the creation, evolution, distribution, synchronization, documentation, and access—all distributed—of knowledge fragments. This seems clear to us based on our current experience, even with our small knowledge base of about 1000 small files.

- We envision knowledge bases exceeding 10K+ files (an order of magnitude bigger) within a ten year time frame. Much of this growth will likely come from stocking the knowledge base with more code and process templates.

4.3 KB Installation, Customization, and Maintenance

- Experienced users can *install* and superficially customize our knowledge base in less than a day. In contrast, extensive customization that involves automated build systems, synchronized repositories, and local contexts may take several days.

  The main point here is that knowledge bases that model complex problem domains are not trivial to install, configure, and test. This realization was unintuitive to us for a long time, because we always performed incremental upgrades to the knowledge base in our development environment. We were not exposed to the full complexity of new installations for a few years. However, when we started to perform multiple installations and customizations at new customer sites, we were soon cured of our naivety in this regard.

- Experienced users can *upgrade* our knowledge base with hundreds of local site customizations within a day at the time of this writing. From this we conclude that incremental upgrades are usually straightforward, and are insensitive to the size of the underlying knowledge base.

- Version control and automatic reinstallation support is almost mandatory in order to properly manage various personal/team/site overrides. This requirement is driven by customization diversity. Hundreds of such overrides can exist for each site, or for each developer that must construct various program test environments.

- Our knowledge base uses mostly ASCII files to hold information. For this reason, customizations are best done by cloning and modifying whole files. This method allows administrators to use simple file-level **diff** operations to search for changed information during upgrades.

- We have learned that updating parts of the knowledge base that are shared by multiple programs must be managed simultaneously with *all* of the multiple programs that use the knowledge. The underlying problem is that programs evolve at different rates, and are released on different release schedules. This means that old shared KB information released with old programs can unintentionally overwrite new shared KB information required by new programs. In such cases, the installation and upgrade processes become order-sensitive.

4.4 General KB Administration

- Incorrect knowledge base configuration is—by far—the primary source of customer trouble reports with our software productivity infrastructure. This understanding was non-intuitive to us, because we thought the organization, content, and use of the knowledge base was relatively straightforward.

  However, this thinking was naive. For several years, we never had to go through a complete reinstallation and customization of the knowledge base at our development site. Thus we were never exposed to the full complexity of customization. Instead, our development knowledge base was incrementally grown over a significant time period.

  In contrast, customers at remote customer sites were obligated to perform repeated full installations and customizations as they deployed the infrastructure tools into new environments and upgraded their knowledge bases every six months or less (to fit our software release cycle).

  The implication of this experience is that for our programs and knowledge base, the knowledge base itself is more sensitive to user-driven failure than are the software programs that use the knowledge. This means that proper system operation is now largely dependent on the customer knowledge base administrators, rather than upon our own internal quality assurance processes.

4.5 KB Access And Performance

- In our problem domain, we have learned that lookup requests are not always simple atomic operations.

  We thought we could use a 5-part key to get a lookup value back, for essentially all lookups. Instead, we found out that knowledge-using programs frequently obtained a filename through a simple key lookup. Then the programs would search that file (and other related files) several times in order to obtain the knowledge required for the task at hand.

  In the end, we identified six distinct types of lookup operations, and ended up with a generic lookup request structure that abstracted all lookup types. Programs now load the lookup structure with the total set of lookup requests they will need to complete their computations.

  Next, the programs issue a single bulk lookup request (perhaps to a remote knowledge server) to obtain the required information. Finally, they begin their computations, performing lookups against their own lookup data structure as the computation proceeds. This method offers acceptable knowledge base performance by avoiding repeated turnaround times with remote knowledge servers.

  The main lesson for us here was that knowledge usage patterns (which we thought were simple lookups) are not always simple in this problem domain. Instead, knowledge searches often involve sequences of lookups (lookup chains), lists of return values (multi-part returns), or dynamic interactions with local program data as lookup sequences proceed.

- We found that file-level granularity of remote lookups was best for most situations in our domain. This choice was driven by issues of repeated access to the same file, simplicity of caching whole files versus individual 5-part key-value pairs, and by implementation complexity.

- Accessing remote knowledge bases over TCP is the only effective way of sharing of large repositories of knowledge. It is obvious that storing all software reuse knowledge (templates, code fragments, process fragments, ...) on one local system will be impossible. Our experience strongly suggests that factors of size constraints, access loads, and knowledge evolution rates all argue against a central local server knowledge model. Instead, a distributed, web-like model (with local caching) is much more likely to be a better long-term solution.

4.6 KB Benefits

- Our experience shows that, in our problem domain at least, a knowledge base is an excellent way of implementing site policy in a non-intrusive way within a software organization. One site administrator configures the system, and all users benefit at essentially no cost.

- Motivated users can override the default site settings by learning the kb mechanisms, but we have found that in general this is rare for users. This suggests that the default knowledge configurations created by site administrators are usually sufficient to meet the needs of most users.

In summary, our experience has convincingly demonstrated that intelligent, knowledge-based systems do actually deliver high-value, low-cost benefits to large user populations without imposing significant knowledge burdens on system users.

5 The Knowledge Base

This section describes the knowledge base: organization, contents, granularity, extensibility, and referencing mechanisms.

5.1 KB Contents

Our knowledge base consists of about 1000 (nearest order of magnitude) small ASCII files that support software engineering and reuse activities in an Internet work environment.

The knowledge base supports about 100 software programs in the infrastructure toolset. About 30 of the programs use the knowledge base in meaningful ways.

Major sections of the knowledge base are concerned with:

1. An automated makefile generation system.
2. An Internet version control system.
3. An Internet automated build system.
4. An Internet software distribution system.
5. A software template instantiation system.
6. Customizations of knowledge in all these areas.

5.2 KB Organization

The knowledge base is organized into 5 levels of increasing specialization:

1. **Search Rules.** All knowledge lookups follow a set of search rules that specify an ordered set of places to look for information. Each place to look is called a *configuration tree*, and contains instances of all the levels described below.

2. **Contexts.** A context is an ordered set of configuration trees. The *initial context* (constructed at system installation time) is used to look up the value of the current context. The value of the current context is a list of configuration directories (that is, a list of search rules) which is then used to complete the knowledge lookup.

3. **Virtual Platforms.** Virtual platforms are named subdivisions of contexts. A virtual platform name is a user-defined subdivision name used to achieve optimal knowledge sharing among platforms in a multi-platform Internet environment.

   Virtual platforms do not have to exist in real life. Instead, sequences of virtual platform names can be chosen to fit the information sharing needs of your particular situation. We use a hierarchy of virtual platform names that progresses from least platform dependent (pi= platform independent) to most platform dependent (sun414 = SunOS 4.1.4). Examples of our virtual platform hierarchies include these: (pi, dos, nt40), (pi, unix, aix, aix420), (pi, unix, sun, sun414), and so on.

   Lookups proceed, in order, from most platform specific (most specialized knowledge) to least platform specific (least specialized knowledge) in the virtual platform tree. Virtual platforms are a major mechanism for handling diversity in the knowledge system.

4. **Program Directories.** Each software program that uses the knowledge base typically has its own subtree of information inside the database.

   Further, there is a distinguished subtree (called the "common" pnode) that holds information shared among multiple programs. In some cases, programs do not have their own program directories, since the programs use only shared information from the common pnode.

5. **Named Lookup Keys.** Each program tree can contain an arbitrary number of knowledge files, organized into an arbitrary number of subdirectories inside the program tree. The host program is the only piece of software that understands the information content and structure of the program tree.

   Each program tree contains a distinguished knowledge file with the same name as the program, extended with the suffix "*.cfg*". All lookups within the program tree start with this file.

   The distinguished configuration file contains named lookup keys that can return values directly, or that can point indirectly to arbitrary collections of other knowledge files.

In some cases (notably for the **getmakes** makefile generation tool and the **stub** code template instantiation tool), the program directories contain tens of subdirectories with tens of additional configuration/template files in each subdirectory. This kind of structure implements additional levels of specialization, from level 6 onwards. We have not yet found a need to go beyond 6 levels of hierarchy in our knowledge base, even in these special situations.

6 Conclusion

We have constructed an intelligent, knowledge-based infrastructure that effectively supports software engineering and reuse in an Internet work environment. The problem domain modelled by the knowledge base—automated, distributed, software reuse involving millions of software components—is a complex problem with much inherent scale and diversity.

We feel that the knowledge base design presented here is a good knowledge model for problems such as the one addressed here. The knowledge base is Internet-scalable, user-extensible, highly customizable, and relatively easily to understand and to maintain.

Future research will likely concentrate on these things:

1. Adding more code templates (passive knowledge).
2. Adding more process templates (active knowledge).
3. Reducing the granularity of knowledge fragments.
4. Developing tools to automatically assemble reduced-granularity knowledge fragments into larger software programs.