Self-Updating Software
9807-15

Progress Report: January 1, 2000—June 30, 2000

Barbara Liskov and Daniel Jackson

Project Overview

In the self-updating software project, we are developing an infrastructure for software that can install, upgrade, and reconfigure itself dynamically, with minimal user intervention. Such an infrastructure might be used for a network of distributed embedded devices that have no user interface and must be updated remotely; for automatic installation and upgrading on personal machines; for maintaining an object database (eg, in a computer-aided design system) across software upgrades; for bringing applications to handheld devices in a just-in-time fashion, and so on.

Our work has focused on two aspects of the problem. The first addresses the problem of updating individual objects in the face of changes to the code of their classes. The second addresses the problem of installing and reconfiguring a collection of software packages in response to requests from a user and notifications of new versions.

In the last six months, we have broadened the scope of the project. Recognizing the centrality of naming to such a scheme, we collaborated with Balakrishnan's group (also funded by this collaboration, under NTT-MIT project 9807-04) to work on the design of his Intentional Naming System. Using our Alloy language and analyzer, we modelled and analyzed the lookup operation, and exposed a variety of design problems that we are now working to resolve.

Although we plan to continue working in this general area, we have not applied for further support. This report therefore marks the completion of the NTT-MIT funded project.

Progress Through June 2000

A. Updating Individual Objects (Liskov)

We have developed a scheme that allows upgrades at a very fine granularity. Suppose that some subset of the classes of a large object-oriented program are to be replaced by new versions. Some of the persistent objects will no longer be usable, since their representations may have changed. Our system therefore automatically locates

objects that need upgrading, and applies a translation to convert their representations. In order to achieve scalability to large object stores, the translation is performed in a lazy manner, so that the cost is not paid until the new version of the object is needed.

A fundamental problem in such a scheme is compatibility of upgrades. Objects are linked together in an elaborate graph; a change to one object may require the upgrading of other objects that reference it. We have solved this problem by introducing a notion of 'complete upgrades'. Before the upgrade begins, we check that the result of applying it uniformly will be a consistent object store; if the check fails, no upgrades occur at all. We believe that it should be possible to perform this check entirely automatically by employing a static analysis of the code.

An upgrade includes the code for transforming individual objects from their old to their new representations. A difficult problem arises when the state of one object depends on the state of other objects. Such dependences may be circular, so it may not be possible to find an order of transformations that would allow us to assume that any objects an object depends on have themselves already been transformed. To avoid this problem, we have developed the notion of a set of 'base methods'; by ensuring that these methods are retained across upgrades we can eliminate any required orderings amongst the transformations of individual objects.

A detailed design of this scheme has been completed. This design is described in Shan Ming Woo's thesis, which was completed in January. Our NTT visitor, Michiharu Takemoto, has developed a variant of Woo’s scheme for a CORBA-compliant ORB, in which each site is managed by a factory that handles the replacement of objects at that site.

B. Updating Configurations (Jackson)

We have built a prototype upgrading infrastructure for software packages.The key idea is that configurations are constructed (and upgraded) automatically from the dependence relationships amongst packages. Each package carries a manifest that indicates which specifications it specifies and which specifications it requires for its subpackages. When a need for a package arises, the specification (currently just a name) is handed to a server, which supplies an address (currently a URL) at which the package may be located. The package is downloaded, and its manifest is analyzed: first to determine that it meets the specification as required, and second to find the packages on which it depends.

A local database on the machine at which the installation is being performed tracks which packages have been installed, and which packages are relying on them. When a need arises, the system actually searches the local database before attempting to download the component. The resulting infrastructure has several key advantages over existing systems, including: fully automatic installation; avoiding multiple downloads and installations of the same component; and the ability to "garbage collect" and remove packages that are no longer needed.

We have demonstrated our prototype both on Java programs, in which the separate packages of the source code are treated as distinct packages, and on a Windows application consisting of binary executables. We extended our model to handle the configuration of packages that have several clients (whether human users or other packages) that customize them in different ways, but did not implement it.

C. Analysis of Intentional Naming (Jackson)

In an *intentional naming* and *resolution architecture*, applications describe their intent and specify *what* they are looking for but not *where* it is situated. This shifts the burden of resolving 'what is desired' to 'where it is' from the user to the network infrastructure. It also allows applications to communicate seamlessly with end-nodes, despite changes in the mapping from name to end-node addresses during the session.

The *Intentional Naming System* (INS) is a framework that provides this functionality, recently developed by Balakrishnan (under NTT-MIT project 9807-04). It comprises applications (clients and services) and *intentional name resolvers* (INRs), which respond to queries from clients. Like IP routers or conventional name servers, name resolvers route requests from clients seeking services to appropriate locations, using a database that maps service descriptions to their physical network locations. But in a name resolver, a service is described using a tree-like structure of alternating levels of attributes and values where an element at a certain level specializes the ones above it.

We studied the fundamental operation, *lookup,* in which an intentional query is mapped to a collection of locations. The structure of queries and the database were modelled in our Alloy language, along with a description of the lookup operation. We used our automatic analyzer for Alloy to investigate the operation, by checking whether all possible executions of the operation that involve queries and databases of some limited size (eg, trees no more than 5 levels deep) satisfy some expected properties. We found a variety of problems. The most interesting involved the implicit treatment of missing attributes as wildcards. Our analysis showed that this could result in the loss of a vital monotonicity property: that after adding a new service to the database, a query that previously returned a location (to an existing service added before) might no longer do so.

A paper describing this work will be presented at the Automated Software Engineering conference in September.

Our model of INS lookup was less than one tenth of the size of the code, and exposed issues that had not been discovered in a year of execution. All analyses terminated in less than 30 seconds, despite covering up to 10^20 cases. We are now working with Balakrishnan to develop a general model of intentional naming, and to resolve the problem of missing attributes.

Research Plan for the Next Six Months

The project has now terminated.