Self Updating Software
9807-15

Progress Report: July 1, 1999 December 31, 1999

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.

Progress through December 1999

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.

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 are now extending our infrastructure to handle the configuration of packages that have several clients (whether human users or other packages) that customize them in different ways.

Research Plan for the Next Six Months

In the next six months, we plan to complete the implementation of our fine-grained scheme for object upgrade and to evaluate it in a case study. This work will be done by Mr. Takamoto, in collaboration with Shan Ming Woo and Miguel Castro. In addition we are planning to write up the work in a paper and submit it for publication.

A fundamental problem in self updating, which we have not yet addressed, is how to determine that components actually satisfy their specifications. Such a check is essential, whether performed online during updating, or in advance.

We have recently developed a new method that can analyze code for conformance to high level design properties. It involves a form of abstract testing, in which a boolean SAT solver is used to simulate a huge number of executions of the component. So far, we have demonstrated that our technique can find flaws in list-manipulation code of the sort usually taken as a challenge for shape analysis. Unlike shape analysis, our technique extends beyond conventional properties (such as lack of sharing) to arbitrary specification properties, and is less prone to spurious error reports. We plan to develop this new scheme, and show that it can be used for rapid assessment of candidate components.