System Support for Adaptation and Composition of Applications

Research abstract for the EuroSys 2008 Doctoral Workshop

Stephen Kell
Stephen.Kell@cl.cam.ac.uk

1 The Problem

A fundamental role of the operating system is to enable separate pieces of software to communicate. It provides the basic communication mechanisms which knit together the complex masses of abstraction logic bridging between hardware- and application-level code. With the increasing ubiquity and diversity of hardware (including mobile devices), the continuing convergence of applications (including telephony and media), new application domains (e.g. social networks, location sensitivity) and ever-more heterogeneous application delivery (e.g. as browser extensions or web applications), there is ever greater need for flexible, composable application code.

This, in turn, requires system support for combining independently developed code in unanticipated novel ways, where these combinations may be specified dynamically by the user. Traditional mechanisms and practices fail to meet these demands in three key areas.

1. System-level communication mechanisms are too many in number and too low-level in nature. Untyped, unstructured communication forces developers to re-invent countless mutually conflicting encodings for the abstract structures and meanings present in the application domain. This, in turn, complicates composition.
2. The operating system's notion of application structure is too coarse-grained. Individual programs, processes or libraries don't adequately identify the boundaries of applications' requirements for isolation, security and communication. Recent research into "browser OSes" and isolation between web applications addresses a special case of this problem [2, 8].
3. There is no convenient support for *interposing* on the communication between separate units of software. Independently developed code is inevitably not plug-compatible; consequently, composability entails support for *adaptation logic.*

Although there are traditional means of performing adaptation for each OS communication mechanism (e.g. wrapper scripts, wrapper libraries, proxies, conversion tools, pipeline filters), these are inadequate. The multiplicity of communication mechanisms means that, for example, an adapting wrapper script is useless if the target application is written to use sockets. Coarse grain means that the desired point of interposition is often unavailable, having been elided at link time. Finally, the byte-based interface at any available point of interposition necessitates tedious adaptation techniques such as regex-based string rewriting.

As a result, application composition is poorly supported by today's OSes. Plug-in and extension systems have provided composability inside the boundary of a single application, but the Unix dream of widespread inter-application cooperation has yet to materialise. To illustrate, here are several simple use-cases which are currently impracticable even for most moderately advanced users.

- sharing bookmarks or history logs across multiple web browsers, or with other classes of application;
- adding a button to invoke a web-based natural language translator from within an e-mail client;
- writing a script (e.g. a `make` rule) which invokes the Postscript generator of an interactive graphical document editor;
- sharing a calendar between multiple applications concurrently, each notifying others of updates;
- pausing a media player application whenever another process requests the sound device.

For the same reasons, re-use of existing code is unnecessarily difficult. Even in the open-source world, where code is freely available, separate projects emerge implementing near-identical functionality, distinguished only by implementation details (choices of windowing toolkit, desktop environment, programming language, storage abstraction, and so on). The motivation for such duplication is invariably that the added homogeneity aids integration with other applications.

My research goal is to create an adoptable system for convenient, dynamic composition of independently developed application code, using *adaptation.*

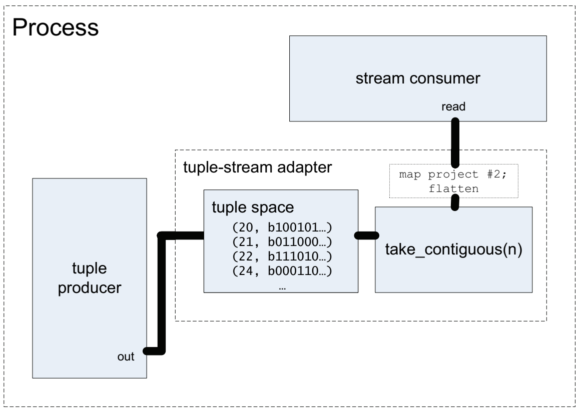
2 The Approach

My approach exploits the following observations.

- In code, there is a desirable but seldom achieved separation of *functionality* from *integration.* The "baking in" of separable details about communication mechanisms and conventions causes mismatches, which make composition harder. We would like to enable and encourage the separation of these concerns, by rethinking the design of linking, loading and IPC mechanisms.
- Scripting languages are popular for "glue code": they have convenient support for many IPC mechanisms, syntactic brevity makes scripts easy to modify *invasively*, and string rewriting features enable adaptation. However, these low-level features are error-prone. We would like to work with a more abstract model.

- *Configuration languages* (ranging from linking languages up to architecture description languages) enable our separation, providing a domain convenient for resolving mismatches. We would like to extend such a language with convenient and powerful support for adaptation.

I am working on the design and implementation of a configuration language, based on the linking language Knit [7] but supporting an extensible set of adaptation primitives (including various published algorithms [9, 6, 1]), custom linkage standards (including e.g. Java binaries), and a higher-level data model. To exploit pragmatic *invasive modification* without discouraging modular re-use, the language supports inline "ad-hoc" adaptation but also *hierarchical* modularisation of re-usable sub-configurations. To illustrate, the following toy example shows adaptation from tuple- to stream-based communication; the code is Knit-like.



```
unit myTupSpc {
 exports [ prod { (int, bit list ) in_lowest () },
           cons { void      out(int, bit list ) } ];
  files { obj_elf ("C", tuplespace.o) } }

unit myTupProd {
 imports [ dest { void outp(bit list , int) } ];
 exports [ /* ... */ ];
  files { obj_elf ("C", tupleprod.o) } }

unit myStreamConsumer {
 imports [ streamProvider { int read(byte addr, int) } ];
 exports [ /* ... */ ];
  files { obj_elf ("C", streamcons.o) } }

unit takeContig {
 imports [ source { (int, bit list ) in_ord() } ];
 exports [ listProvider { (int, bit list ) list get() } ];
  files { obj_elf ("C", take_contig.o) } }

/* The definitions above could be auto-generated from C code.
 * The definition below links them using ad-hoc adaptation.
 * Note differing symbol names and argument orders, and the
 * 'map' and 'project' used to turn tuples into a string. */

unit Process {
 exports [ /* ... */ ];
 link exec_elf ("process") {
  myTupProd.dest <- myTupSpc.cons { outp(a, b) <- out(b, a) }
  takeContig.source <- myTupSpc.prod { in_ord <- in_lowest }
  myStreamConsumer.streamProvider <- {
   read <- flatten(map (project #2),
     takeContiguous.listProvider.get)
}}}
```

To address dynamism, I will embed a variant of this language into the operating system's dynamic loader. This subsumes of a host of other IPC interfaces while supporting dynamic, user-directed adaptation. (Currently `dlopen()` accepts a library name; this extends to an adaptation expression in the variant language. Likewise, library handles and file handles may be unified.) As with LLL [5], the structure of running applications will be dynamically manipulable, but with the added benefit of adaptation support.

3 The Plan

Current implementation work is extending Knit with the new features. Early case-studies will capture the linkage graph of a large existing codebase, such as Firefox, then produce new compositions of its features with external codebases (e.g. porting the bookmarking feature to a file manager) and develop a small library of common adaptations. The library can exploit the pre-existing selection of "packagings" [3]: it suffices to adapt to and from a variety of known linkage forms.

Later work will tackle the dynamic case by extending Linux's dynamic loader, and demonstrate how this subsumes traditional IPC (e.g. mapping file handles to object handles; objects need not be simple files). With help from the library, users can dynamically specify extensions to their running applications by combining generic third-party code with adaptation logic. A possible complement is a refactoring tool, for human-assisted separation of integration from functionality within existing C code (suitably preprocessed [4]).

Evaluation will proceed by both case study and software measurement. Separating functionality from integration aims to reduce the complexity of adaptation, i.e. to reduce *local coupling*. Existing measures are ad-hoc and based on models which do not feature adaptation. I propose a new measure grounded in information theory, in which local coupling is related to the combined entropy of module interface specifications and the adaptation logic required to bridge mismatches.

Implementation is only just beginning, but I hope to have useful results by the time of the workshop.

References

[1] A. Bracciali, A. Brogi, and C. Canal. A formal approach to component adaptation. *J. Sys. Soft.*, 74:45–54, 2005.

[2] R. Cox, J.G. Hansen, S. Gribble and H. Levy. A safety-oriented platform for web applications. *Proc. IEEE. Symp. Sec. Priv.*, 2006.

[3] R. DeLine. Avoiding packaging mismatch with flexible packaging. *IEEE Trans. Soft. Eng.*, 27:124–143, 2001.

[4] B. McCloskey and E. Brewer. Astec: a new approach to refactoring C. *Proc. 10th ESEC / 13th FSE*, 2005.

[5] J. Mukherjee and S. Varadarajan. Develop once deploy anywhere: achieving adaptivity with a runtime linker/loader framework. *Proc. 4th Workshop on Reflective and Adaptive Middleware Systems*, 2005.

[6] R. Passerone, L. de Alfaro, T. Henzinger, and A. Sangiovanni-Vincentelli. Convertibility verification and converter synthesis: Two faces of the same coin. *Proc. Int. Conf. Computer-Aided Design*, 2002.

[7] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit: Component composition for systems software. *Proc. 4th OSDI*, pages 347–360, 2000.

[8] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and communication abstractions for web browsers in MashupOS. *Proc. 21st SOSP*, 2007.

[9] D. Yellin and R. Strom. Protocol specifications and component adaptors. *ACM TOPLAS*, 19:292–333, 1997.