Configuration and Adaptation of Binary Software Components

Stephen Kell
Computer Laboratory
University of Cambridge
15 JJ Thomson Avenue, Cambridge CB3 0FD, UK
Stephen.Kell@cl.cam.ac.uk

Abstract

Existing black-box adaptation techniques are insufficiently powerful for a large class of real-world tasks. Meanwhile, white-box techniques are language-specific and overly invasive. We argue for the inclusion of special-purpose adaptation features in a configuration language, *and outline the benefits of targetting* binary *representations of software. We introduce Cake, a configuration language with adaptation features, and show how its design is being shaped by two case studies.*

1. Introduction

Under decentralised development, it is inevitable that most code will be written independently of most other code with which it could usefully be combined. We therefore require tools for realising compositions of *independently evolving* code, and even completely *unanticipated compositions* of code. The key challenge of such compositions, beyond the traditional challenge of "programming in the large" [6], concerns overcoming *mismatch* between components' interfaces.

Conventional approaches to mismatch are ad-hoc: most common is invasive editing of source code. This yields a brittle patchset that is expensive to maintain. Aspects and other white-box techniques have been proposed to remedy this [8], but such invasive techniques are language-dependent and arguably damage modularity [16]. A less powerful but more modular approach is to manually code a black-box adaptor: this avoids dependence on source code details, by expressing adaptation relative to an interface definition only. Although limited to interface-level adaptations, and still fragile to the extent that interface details change, adaptors have proved useful both as a design pattern in conventional languages [9] and as a domain for specialised tools and languages [13, 17, 10, 7]. We embrace the latter approach, but target its practical weaknesses thus far.

The most visible decentralised development effort is in the open-source community. Of the many useful research tools developed to enable adaptation, none applies effectively to a substantial range of the software produced by this community. C and C++ remain popular languages for large projects, whereas the many adaptation tools targetting Java and other higher-level runtimes [10, 7] sidestep a large class of mismatches concerning memory allocation and pointer use. Several common classes of mismatch arise in practical adaptation tasks, but many tools focus only on one or other of argument- [13], protocol- [17], or wiring-level [14] adaptations, so apply to few real use-cases. Another common source of mismatch lies in the parallel reimplementations of various recurring abstractions (e.g. objects, common data structures, RPC, many-to-many communication); some tools support interchange among fixed sets of these [3, 5], but none supports *description* of an *open* set of these.

This paper describes our ongoing work towards the following goals:

- to create a high-level language for describing primarily black-box adaptations, which can apply to components written in various languages including C and C++;
- to ensure a language design sufficiently expressive to apply to a wide class of real-world adaptation tasks;
- to demonstrate the feasibility of performing *link-time* adaptation on fully-compiled *binary* components.

2 Overview

We are developing Cake, a language for high-level description of adaptations. Its goal is to reduce the practical complexity of performing real-world adaptations; it does so by borrowing ideas from various *configuration languages*. These are a spectrum of languages explored in mostly parallel strands of research, from low-level linking languages [14] through module interconnection languages [6], coordination languages [1] up to high-level

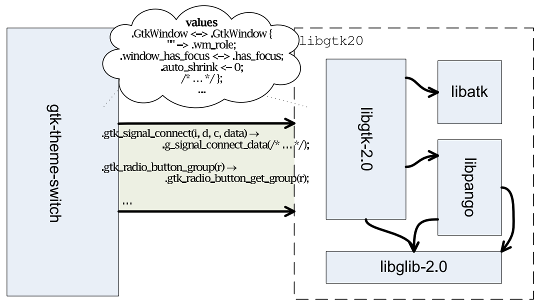

values
.GtkWindow <–> .GtkWindow {
"" –> .wm_role;
.window_has_focus <–> .has_focus;
.auto_shrink <– 0;
/* ... */ };
...
libgtk20
gtk-theme-switch
.gtk_signal_connect(i, d, c, data) → .g_signal_connect_data(/* ... */);
.gtk_radio_button_group(r) → .gtk_radio_button_get_group(r);
...
libgtk-2.0
libatk
libpango
libglib-2.0

Figure 1. Configuration with adaptation logic

architecture description languages [12, 15]. All share a key property: they mitigate complexity by capturing *inter-component relationships*, such as communication topologies, information-hiding or parallel execution constraints. Since adaptation requirements are also a function of these relationships (specifically of asymmetries between *provided* and *required* interfaces), we believe that adaptation logic is best captured within a configuration language. Unsurprisingly, existing configuration languages often enable limited forms of adaptation—e.g. by interposing conversion modules, modifying synchronisation conditions, or replacing communication subsystems—but are neither powerful nor complete enough to be practical as adaptation tools.

Cake targets binary representations of software (specifically relocatable ELF object files at present). This has several advantages: binary formats unify multiple source languages, so have wide applicability; adaptation can conveniently be performed *late* on code already deployed; finally, it works when source code is unavailable, provided that the binary can be comprehended by other means (such as documentation, metadata or reverse-engineering tools).

Object code presents several challenges: there are few constraints on the use of memory, and often no run-time safety guarantees or metadata. This widens the space of adaptations required, and complicates automated analysis. However, recent work has shown binaries to be surprisingly tractable, witnessed by the success of link-time optimisers [4], binary instrumentation tools [11], binary-rewriting virtual machine monitors[1] and reverse-engineering tools [2].

Another difficulty is that programmers must now deal with two different views of their code: the source view and the binary view. These may differ: firstly from interprocedural optimisations (e.g. inlining); secondly from name-mangling and other unfriendly encodings. These can be mitigated: the former by delaying interprocedural optimisation until link-time [14, 4], and the latter with tool support for describing *interpretations* of such encodings (see §3.4).

[1]For example VMware, http://www.vmware.com/.

3 Design and case studies

To prototype the Cake language, we are conducting case studies where we implement glue logic conventionally, then devise Cake syntax expressing that same logic. This paper discusses two case studies. The first examines mismatch across major *evolutions* of interfaces, specifically in the Gtk+ family of libraries.[2] We took the gtk-theme-switch[3] client, available in two separate source forks for API versions 1.2 and 2.0, and adapted the binary 1.2 client to link with the 2.0 libraries. The second case study examines *unanticipated composition*: we took the rox-filer file manager[4] and replaced its simple built-in history log (which remembers visited directories) with the history from the Konqueror web browser.[5]

3.1 Introducing components

Statements in Cake are of two main kinds: about components which already *exist*, and about ones which Cake must *derive*. The simplest derivation just invokes the linker.

```
  exists  elf_archive ("libgdk−x11−2.0.a") gdk−x11−2.0;
  exists  elf_archive (" libatk −1.0.a")  atk−1.0;  //  ...  more follow
  derive  elf_reloc_archive ("gtk−libs−2.0.a")  libgtk20
      = link [gtk−x11−2.0, gdk−x11−2.0, atk−1.0, /∗ ...  ∗/];
```

Cake derives glue logic using information from three sources: programmer annotations in the input file, metadata in object files, and static analysis. It treats these sources interchangeably: for example, if an input file lacks certain metadata, this may be provided as annotations; static analysis may infer certain annotations omitted by the programmer, or verify ones provided. Programmer knowledge is invaluable in creating efficient and maintainable adaptors. For example, a programmer may know that a field in one library's data structure is ignored by the intended client; therefore, adaptor logic between the two need not specify how to represent this field on the client side. Annotations such as this appear in a component's exists statement, and are subject to controllable checking using keywords check (raises an error if metadata or analysis cannot verify the annotation), declare (allows unverifiable annotations, but flags contradictions as errors) or override (unconditional).

```
exists  elf_archive (" libgtk −x11−2.0.a") gtk−x11−2.0 {
    override { . gtk_dialog_new  :  _ → GtkDialog ptr } };
//  ...     ˆ−− override the imprecise type found in debug info
/∗ Declare the client binary, adding annotations. ∗/
exists  elf_reloc ("switch.o")  switch12 {
  declare {  . gtk_dialog_new  :  _ → object { //  this component
                 .vbox: opaque ptr;  //  treats 'vbox' opaquely
                 _ : ignored } ptr }//  and ignores other fields
  /∗ more annotations  ...  ∗/ }     //  on the returned object
```

[2]http://www.gtk.org/
[3]http://www.muhri.net/nav.php3?node=gts
[4]http://rox.sourceforge.net/
[5]http://www.konqueror.org/

3.2 Function correspondences

Adaptations in Cake are expressed using *pattern-matching* and *correspondences*. For example, the following correspondence

```
switch12 ↔ libgtk20 {
  . gtk_signal_connect (i, d, c_h, data)
    → . g_signal_connect_data (i, d, c_h, data, null, {}); }
```

states that a call to .gtk_signal_connect in component switch12 corresponds to a slightly different call in libgtk20. The left-hand side constitutes a pattern whose matched elements are instantiated on the right of the arrow. Most arrows are bidirectional, implying a symmetric correspondence. Unidirectional arrows indicate correspondences which only apply in one direction—these are useful to describe asymmetric correspondence rules. (Unidirectional arrows also appear in value correspondences, to provide default values for missing fields—see §3.3.)

More complex patterns are useful. Sequences of calls can adapt cases where one function call has become two (e.g. the splitting of gtk_text_new into gtk_text_buffer_new and gtk_text_view_new) or vice-versa.

```
switch12 ↔ libgtk20 { /* provides / requires correspondences. */
  (t = . gtk_text_new (null, null);  ...;  gtk_text_insert (t, ...))
      →                                   // call sequence pattern
      (tb = . gtk_text_buffer_new (null);
       tv = . gtk_text_view_new (null);  // new call sequence
       . gtk_text_view_set_buffer (tb) ); }
```

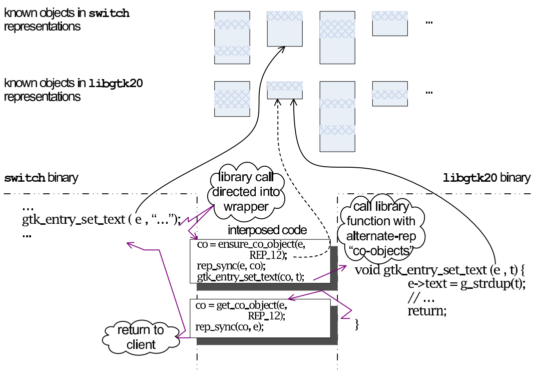
This shows a simple loop-free imperative sublanguage expressing sequences of calls, which Cake compiles into stubs interposed at link-time. Simple local call sequences may be matched statically on the control flow graph. More generally, a dynamic approach is needed, amounting to protocol adaptation [17], although our case studies have so far not required this generality. Similarly, dynamic matching can be used to support patterns which match a call only when certain argument values are passed.

3.3 Value correspondences

Value correspondences capture equivalences between sets of values flowing between mismatched components. The following might appear in a derive block

```
values switch12.GtkWindow ↔ libgtk20.GtkWindow {
  "" → .wm_role; // default value for new field
  .type { names .GtkWindowType } // enum in disguise, so
      ↔ .type { names .GtkWindowType }; // give names
  .window_has_focus ↔ .has_focus;
  . auto_shrink ← 0; /* field removed */ }
```

to give correspondences between values of respective structures (both named GtkWindow), providing default values for missing fields. Name equivalences are drawn by


known objects in switch representations
...
known objects in libgtk20 representations
...
switch binary
libgtk20 binary
library call directed into wrapper
interposed code
call library function with alternate-rep "co-objects"
return to client

```
...
gtk_entry_set_text ( e , "..." );
...
```

```
co = ensure_co_object(e,
                      REP_12);
rep_sync(e, co);
gtk_entry_set_text(co, t);
```

```
co = get_co_object(e,
                   REP_12);
rep_sync(co, e);
```

```
void gtk_entry_set_text (e , t) {
        e->text = g_strdup(t);
        // ...
        return;
}
```

Figure 2. Interposing on object exchange

default between like-named fields, inserting relevant coercions for primitive DWARF datatypes if lossless; the programmer need only specify exceptions to these rules. Name-equivalence can be requested for simple values, using name–value mappings either provided manually as annotations, or by named enumeration types in the debug information. Regular expression correspondences are supported for variable-length data, as in the libkonq case study's conversion between pathname and URL strings.

Objects are structured values with identity and lifetime. In the Gtk+ case study, client and library exchange objects on the heap, and their expectations of object sizes and layouts do not match. To interpose on the exchange of such objects, we identify calls where mismatched objects may be passed (directly or indirectly), and link in code to create and synchronise "logical replica" objects (each with differing layout), much like deep copying in an RPC system. Copy depth is limited using knowledge of which fields are actually examined, e.g. from opaque and ignored annotations (§3.1). Since the client ignores most fields, most pointers need not be followed and most fields need not be synchronised. This also helps avoid "forked" objects, where conflicting updates have been made to different replicas. Special handling is also required when passing function pointers, testing object identity, and in scheduling synchronisation of replica objects. Despite obvious limitations, we are hopeful this approach will prove sufficient for a wide range of practical adaptation tasks.

Getting history data from libkonq to rox entailed interposing on rox's history query function, to rebuild rox's linked list from one provided by libkonq. The above approach applies similarly in this case. However, in both of these case studies the correspondences are simplified by the 1:1 correspondence between objects. Full generality demands consideration of arbitrary graph transformations. In practice tree rewriting syntaxes (with back-edges treated opaquely) may suffice, perhaps similar to CSS selectors [6] or

[6]http://www.w3.org/TR/CSS2/selector.html

XSLT[7]. Prototyping this is future work.

3.4 Further features and future work

Correspondences shown so far have been particular to a *pairing* of components, but others may be *global*, referencing a single component only. In the static case, these are effectively *rewrites* to the input binaries. We found this useful for rewriting embedded constant data (e.g. pathname of the Gtk+ config file), and for replacing function bodies in two cases. Firstly, the behaviour of the libkonq history logger was to ignore file:// URLs; since we wished to log precisely these, we redefined its filterOut function to return true. Secondly, to avoid linking rox with the whole of bulky libkonq.a, we discarded all but the konq_historymgr.o component and its immediately depended-upon functions, then replaced the latter with stub implementations.

In the Gtk+ case study, another mismatch is in the data read from config files: the file format differs between library versions. A dynamic sequence of fread() calls can be *interpreted* as a *higher-level channel*; given a suitable grammar, the same tree-based rewriting techniques applied to objects (above) could be used to transform this file data.

Unexpected complexity came from communication paths in libkonq: calls to add history entries are not simple function calls, but are indirected through the X server and a DCOP server process, then dispatched by a separate listener thread. Most of the adaptor's complexity was in correctly initialising both the listener and the DCOP-layer state for routing these messages. DCOP is a well-defined style of *packaging* [5]; knowledge of it should be expressible in Cake, from which code to support any binding of a regular C function (as in rox-filer) to a DCOP handler (in libkonq) could be generated.

Although not encountered in our case studies, *memory allocation* is a likely source of mismatch: whether callee or caller is responsible for freeing memory, when this may happen, what mechanism is used, and what accounts must be kept (e.g. reference counts). Suitable annotations (on pointer arguments) and packaging-style descriptions should allow concise description of the necessary adaptation.

In both of our case studies we were fortunate that our task was *well-abstracted*: linkage boundaries exposed the necessary points of interposition. On occasions where this is not true, we must fall back on instrumentation [11] and other more invasive techniques to recover those points.

4 Status and acknowledgements

Ongoing work is generalising the manual glue code produced during the case studies into code generation logic in the Cake compiler. Further case studies are planned, including application of Cake to binary device drivers. We hope that use of Cake together with a link-time optimiser [4] will show that performance overheads need not be great.

The author is extremely grateful to Michael Hicks for most productive and timely feedback, and to David Greaves for helpful discussions and support.

[7] http://www.w3.org/TR/xslt

References

[1] F. Arbab and F. Mavaddat. Coordination through channel composition. In *Proc. Coordination*, pages 21–38, 2002.

[2] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum. CodeSurfer/x86—a platform for analyzing x86 executables. In *Proc. 14th Intl. Conf. Compiler Construction*, 2005.

[3] J. Callahan and J. Purtilo. A packaging system for heterogeneous execution environments. *IEEE Transactions on Software Engineering*, 17:626–635, 1991.

[4] B. De Sutter, B. De Bus, and K. De Bosschere. Link-time binary rewriting techniques for program compaction. *ACM TOPLAS*, 27(5):882–945, 2005.

[5] R. DeLine. Avoiding packaging mismatch with flexible packaging. *IEEE Transactions on Software Engineering*, 27:124–143, 2001.

[6] F. DeRemer and H. Kron. Programming-in-the large versus programming-in-the-small. In *Proceedings of the International Conference on Reliable Software*, 1975.

[7] D. Dig, S. Negara, V. Mohindra, and R. Johnson. ReBA: a tool for generating binary adapters for evolving Java libraries. In *Proc. 30th ICSE*, 2008.

[8] M. E. Fiuczynski, R. Grimm, Y. Coady, and D. Walker. patch (1) considered harmful. In *Proceedings of the 10th conference on Hot Topics in Operating Systems*, 2005.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[10] R. Keller and U. Holzle. Binary component adaptation. In *ECOOP '98*, pages 307–329, 1998.

[11] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proc. PLDI*, 2005.

[12] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference*, 1995.

[13] J. Purtilo and J. Atlee. Module reuse by interface adaptation. *Software - Practice and Experience*, 21:539–556, 1991.

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

[15] M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21:314–335, 1995.

[16] F. Steimann. The paradoxical success of aspect-oriented programming. In *Proc. OOPSLA*, 2006.

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