Component Adaptation and Assembly Using Interface Relations

Stephen Kell

Computer Laboratory, University of Cambridge
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
firstname.lastname@cl.cam.ac.uk

Abstract

Software's expense owes partly to frequent *reimplementation* of similar functionality and partly to *maintenance* of patches, ports or components targeting evolving interfaces. More modular non-invasive approaches are unpopular because they entail laborious wrapper code. We propose Cake, a rule-based language describing compositions using *interface relations*. To evaluate it, we compare several existing wrappers with reimplemented Cake versions, finding the latter to be simpler and better modularised.

Categories and Subject Descriptors D.2.3 [*Coding Tools and Techniques*]; D.2.12 [*Interoperability*]

General Terms Languages

1. Introduction

Today's software development ecosystem is vast in scale and decentralised in nature. Inevitably, most code is written in isolation from most other code with which it could usefully be combined. Most software grows *upwards* in stacks or *silos*, each piece written "for" some specific infrastructure: libraries *for* some programming language, tools *for* some IDE or editor, plugins *for* some web browser or media player, applications *for* some operating system or desktop suite or hardware platform. Put differently, interface mismatch abounds in software. Since there are always different ways of expressing the same meaning, components that are logically compatible nevertheless evolve with mismatched interfaces. Current software practices fail to exploit the compositional potential within existing code; they encourage from-scratch development and coupled code.

OOPSLA/SPLASH'10, October 17–21, 2010, Reno/Tahoe, Nevada, USA.

The adapter pattern [Gamma et al. 1995] is a noble failure. It provides a modular, compositional approach to mismatch, but despite these potential benefits, programmers faced with mismatch usually choose to *port* source code by invasive editing. This is unfortunate, because much expense in software stems from *many versions of similar things*. Multiplied code means multiplied complexity and multiplied costs. Invasive porting also risks introducing bugs to working code. To avoid these costs, we must make non-invasive adaptation a more effective option for the programmer.

Adapters can be described an order of magnitude more simply than conventional tools allow. Currently, writing adapters is repetitive, error-prone and inconvenient. We introduce a tool for concise, high-level and convenient description of adapters, based on our special purpose language Cake. Cake is rule-based: the Cake programmer declaratively specifies how components' interfaces relate. Cake advances on prior work by supporting context-dependent, many-to-many relations between interface elements, and in its automatic treatment of complex object structures. By targetting binaries, Cake is convenient, and eliminates complexity associated with multiple build environments.

The contributions of this paper are as follows.

A language of correspondences Cake abstracts black-box adaptations using rules called *correspondences*. These both unify and extend the expressiveness found in prior work on black-box adaptation. In particular, they support complex relations between interfaces, of the kind found in real adaptation tasks.

Object structures By applying correspondences while also following pointers, Cake adapts complex object structures at no extra effort. Our implementation, which targets native binaries, makes use of novel well-behavedness criteria in order to correctly discover object structures in real code at run time.

Benefits We show, by using Cake to reimplement some pre-existing wrappers, how Cake results in simpler, better-modularised code.

We begin with Cake's motivation and design goals.

2. Motivation

Currently, non-invasive adaptation is usually eschewed in favour of invasive editing or from-scratch redevelopment. To understand why, consider that adapters currently consist of *wrapper* functions like the two in Fig. 1. These form part of an adapter (covered fully in §6.1) between a pair of filesystem interfaces, puffs (here exported) and rump (here consumed). (The seek call repositions an open file cursor, while remove deletes a directory entry.) We have added comments, but the details are not important. Instead, notice several problems with this style of code.

```
int p2k_node_seek(struct puffs_usermount *pu,
     puffs_cookie_t opc, off_t oldoff, off_t newoff,
     const struct puffs_cred *pcr)
{ kauth_cred_t cred;
  int rv;

  cred = cred_create(pcr); // convert auth token
  VLE(opc);              // lock vnode ptr
  rv = RUMP_VOP_SEEK(opc, oldoff, newoff, cred); // call
  VUL(opc);              // unlock vnode ptr
  cred_destroy(cred);    // destroy temp auth token

  return rv;
}
int p2k_node_remove(struct puffs_usermount *pu,
  puffs_cookie_t opc, puffs_cookie_t targ,
  const struct puffs_cn *pcn)
{ struct componentname *cn;
  int rv;

  cn = makecn(pcn);      // issue temp name
  VLE(opc);              // lock vnode ptr
  rump_vp_incref(opc);   // bump refcount
  VLE(targ);             // lock target vnode
  rump_vp_incref(targ);  // bump that refcount
  rv = RUMP_VOP_REMOVE(opc, targ, cn); // call rump
  AUL(opc);              // this time, vnodes were unlocked
  AUL(targ);             //... by rump, so just assert this
  freecn(cn, 0);         // free temp name

  return rv;
}
```

Figure 1. Example filesystem wrapper code

Repetition A large volume of similar code is required for a conceptually simple task.

Poor modularity The code is not *trivially* repetitive. Each wrapper applies a different subset of rules, e.g. for treatment of arguments. Consider `opc` above: one case requires a bumped reference count and has different unlocking semantics from the other. The programmer must juggle these rules correctly amid the sea of similarity.

Inconvenience Among other headaches, to compile this code the programmer must construct a hybrid build environment supporting compilation against both interfaces.

Complexity This shows a very simple case, where functions correspond one-to-one. In others, complexity quickly escalates.

```
// rules concerning functions
p2k_node_seek(_, vn, oldoff, newoff, cred)
     ⟶ RUMP_VOP_SEEK(vn, oldoff, newoff, cred);
p2k_node_remove(_, vn as vnode_bump, tgtvn as vnode_bump,
 cn) ⟶ RUMP_VOP_REMOVE(vn, tgtvn, cn);

// rules concerning values
values puffs_cookie_t ⟶ ({VLE(that); that}) vnode;
values puffs_cookie_t ⟵({VUL(that); that}) vnode;
values vnode_bump ⟶({VLE(that); // also bump refcount
           rump_vp_incref(that); that}) vnode;
values vnode_bump ⟵vnode; // unlock not required
values puffs_cred (cred_create(this)) ⟶ kauth_cred;
values puffs_cred ⟵(cred_destroy(this)) kauth_cred;
values puffs_cn (makecn(this)) ⟶ component_name;
values puffs_cn ⟵(freecn(this, 0)) component_name;
```

Figure 2. Cake rules generating equivalent wrappers

In short, wrappers are an unnecessarily complex approach to adaptation. Cake is a language designed to fix this problem. Figure 2 shows some Cake rules sufficient to generate the wrappers in Figure 1. Again the details are not important, but notice several advantages.

Separation of concerns The Cake programmer writes rules which we call *correspondences*. Each rule localises a particular piece of domain-specific knowledge about the adaptation task. The compiler is responsible for composing rules into wrappers. In particular, notice here that rules concerning functions and rules concerning values are kept separate. Such rules form the basic Cake language (§3).

Expressiveness Cake rules advance on prior work by supporting *context-sensitive* and *many-to-many* relations between interface elements. For example, a single function may map to one of several calls on the opposing interface, depending on what calls have come before, or to a sequence of calls. Similarly, sets of values or objects occurring together may be treated as a group, and corresponded by a single rule. These and other advanced features greatly extend the power of the Cake language (§4).

Object structures This example passes only isolated objects across the interface. However, Cake can handle the exchange of arbitrary object graphs across mismatched interfaces. This can eliminate considerable code: consider a wrapper walking a linked list to convert each element in turn. The programmer need only specify how separate classes of object relate; the Cake runtime automatically explores the graph, applying rules to the objects it finds (§5).

Simpler, shorter code The rules above may appear to be only a little shorter than the wrapper code. However, the entire p2k adapter contains not two but 28 wrapper functions. Each rule above contributes to *many* of these wrappers, and often many wrappers can be generated from a single rule. The result is shorter, more readable and more maintainable code, as we show in three case studies (§6).

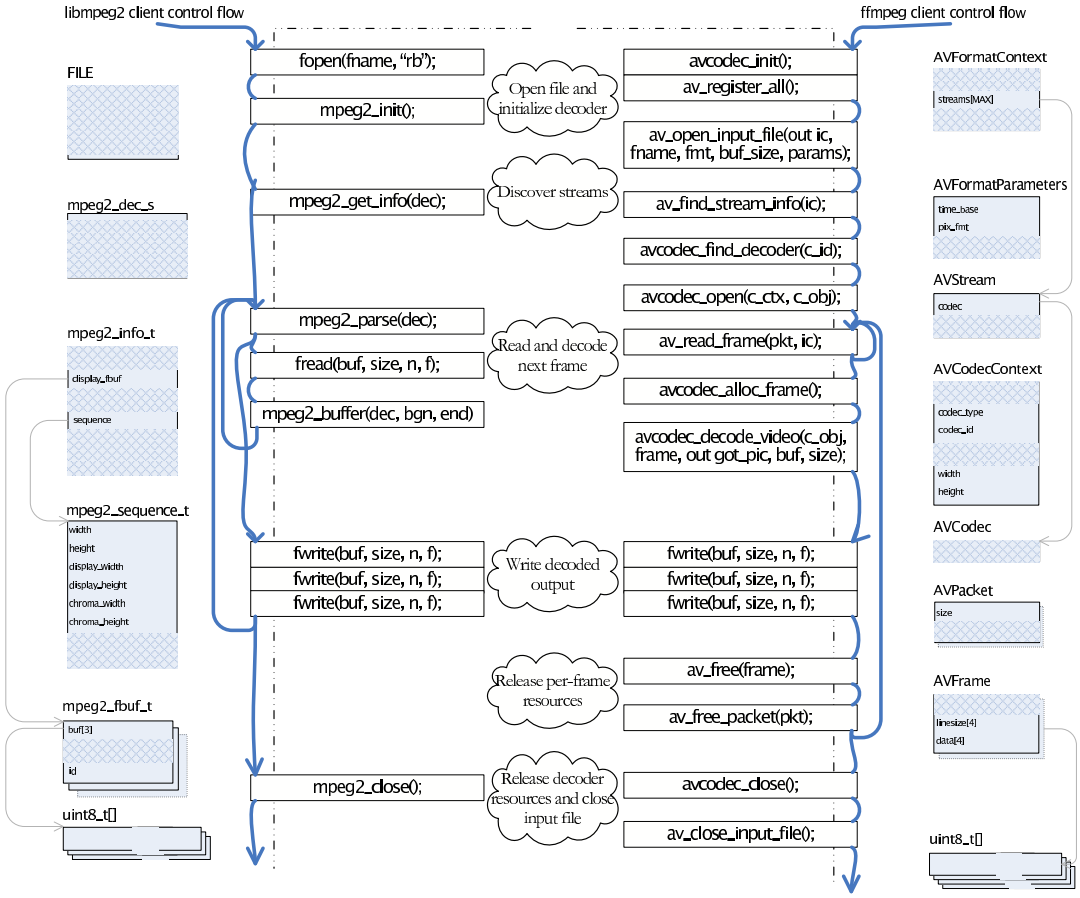

Figure 3. Example comparable usage patterns for libraries libmpeg2 and ffmpeg

3. The design of Cake

We use a relatively ambitious running example to illustrate the design of Cake as a tool and a language. Can we take a client and library implementing hitherto unrelated interfaces and, by writing a succinct description of their correspondences, glue the unmodified binaries together?

Consider a simple program which uses a library to decode some video. There are many possible choices of library; we consider a client written against the libmpeg2 library[1]. Suppose we wish to link this instead against the ffmpeg family of libraries[2]. This has many plausible motivations: perhaps to reduce the dependency footprint of a larger system, perhaps to exploit the larger feature-set of ffmpeg (which can decode video in other encodings than MPEG), or perhaps for differences in reliability or performance. Figure 3 shows equivalent usage patterns of the two interfaces. Note that the correspondence between the two is nontrivial: in most cases there is no one-to-one correspondence between either the objects or the function calls used by the two interfaces.

[1] http://libmpeg2.sourceforge.net/

[2] http://ffmpeg.org/

3.1 Insights

The primary design goal of Cake is to abstract composition tasks in a modular way. In the spirit of Parnas's information hiding [Parnas 1972], one useful approach is to restrict the programmer's attention to some notion of *interface*, just as Fig. 3 describes the two components only in terms of the function calls and data structures that they exchange. We refer to this as *black-box* adaptation (cf. white-box approaches, which may reference arbitrary internals of a component).

A convenient formalisation of this notion of interface is the *trace* of a component's interactions, of the sort displayed by tools such as ltrace[3]. Figure 4 shows the ltrace output for our client's interaction with libmpeg2.so. Abstractly, a trace is simply a sequence of calls or *events*, each communicating zero or more values. Cake code consists largely of rules which, abstractly, describe a *transducer* over this trace—that is, an automaton which both recognises and generates. At run time, Cake-generated code feeds each component a trace generated from those output by the other components. Note that our discussion of traces is purely conceptual; a Cake

[3] http://ltrace.alioth.debian.org/

```
mpeg2_init()       = 0x9cd6180
mpeg2_info(0x9cd6180)        = 0x9cda380
mpeg2_parse(0x9cd6180)       = 0
mpeg2_buffer(0x9cd6180, 0xbfd17d88, 0xbfd18d88) = 0x9cd6180
mpeg2_parse(0x9cd6180) = 1
# --- snipped ---
mpeg2_parse(0x81bf180) = 0
mpeg2_buffer(0x81bf180, 0xbf9e2a58, 0xbf9e2a58) = 0x81bf180
mpeg2_close(0x81bf180) = 1
```

Figure 4. Example trace of a libmpeg2 client

programmer never needs to generate or manipulate traces in any way.

3.2 Requirements

What kinds of rules are required for realistic adaptation tasks like our video decoding example? From Fig. 3 it is clear that simple remappings of function signatures and object fields are not sufficient, for several reasons.

- Correspondences between events are not one-to-one. In ffmpeg there is usually more than one call for each libmpeg2 call, so we require a way of mapping one call to many. Sometimes this relationship is reversed, so we need to recognise a sequence of many calls and map it back to a singleton.
- Arguments to one call may not be sufficient to perform the corresponding call. For example, mpeg2_get_info() maps to av_find_stream(), but the latter needs a reference to the input file—rather than the decoder, which inconveniently is the only argument to mpeg2_get_info().
- Components differ in the shapes of their data structures. Single fields or single objects may correspond to many fields or many objects. Moreover, objects may be passed indirectly, perhaps over many levels of indirection from the arguments themselves.

These imply that our transducer needs to be *stateful*, and that it must be able to navigate object structures. At run time, Cake maintains two kinds of state: *pattern state*, which enables matching of calls in a context-sensitive fashion; and *association state*, which tracks sets of semantically related objects collaborating across sequences of calls. The programmer does not manipulate this state directly, but embodies it in abstract rules. Before describing these rules, we provide a summary of some additional design goals and a high-level view of Cake.

3.3 Additional goals

Our design accommodates several other goals which make Cake a more effective programming tool.

Target binaries Since traces are agnostic to source code and source language, Cake affords the convenience of working on binaries—the form in which software is usually deployed. To apply Cake to some installed software, it is not necessary to reproduce the build environment for that software, or recompile the software, nor even to possess source code for target components.

Applicability We want Cake to apply to a large volume of existing components. We chose (somewhat arbitrarily) to focus on components produced in the open-source community. These are often written in C, C++ and other unsafe languages with explicit storage management. This entails certain memory-aware adaptation features (§4.6), and careful treatment of pointers (§5). However, the core problem which Cake addresses, namely interface mismatch among components, is specific neither to binaries nor to unsafe languages.

Non-goals Our design sacrifices focus on other potential goals. We will touch on reasons why efficient implementation is possible, and why formal reasoning about Cake code is feasible, but these are not explicit goals. Also, despite targetting modularity, note that Cake is not a *module system* per se: it does not define any novel abstraction of components themselves. Rather, it abstracts *differences among* component interfaces; its notions of component and interface are conventional.

Safety At first glance it may appear risky to perform programming tasks at the binary level. We firmly believe that it need not be less safe than any existing source-level approach. For reasons of simplicity, Cake has not initially been designed to provide *guarantees* of safety. However, binaries admit exactly the same sorts of type-checking and local reasoning as source-level representations, given appropriate metadata.

Well-abstractedness We also assume that our task is "well abstracted", meaning that there is sufficient information in traces generated (and traces accepted) to express the required composition. This is precisely the necessary condition for a black-box approach to suffice. We believe white-box techniques (including aspect-oriented programming, instrumentation systems and so on) to be an essential complement to black-box ones, especially for turning *non*-well abstracted tasks into well-abstracted ones, which we consider for future work (see §7).

Computational power Cake is not Turing-powerful. We believe that future work can semi-automate the generation of Cake code (discussed briefly in §7). Cake is emphatically *not* a language for implementing new functionality, so it can afford to sacrifice some computational power for tractability.

Support heterogeneity Software is developed in a multitude of languages and coded in a multitude of styles. We want Cake to embrace this diversity by enabling low-cost mix-and-match of heterogeneous components. Since much code can be compiled down to a single binary representation, Cake is well-placed for this. So far our examples have centred on C-language codebase; many other procedural and object-oriented languages also fit Cake's model and could be supported with little effort (primarily back-end support for

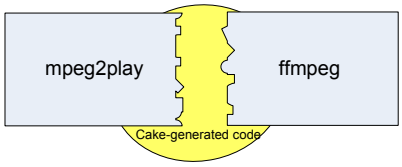

Figure 5. High-level view of an application of Cake

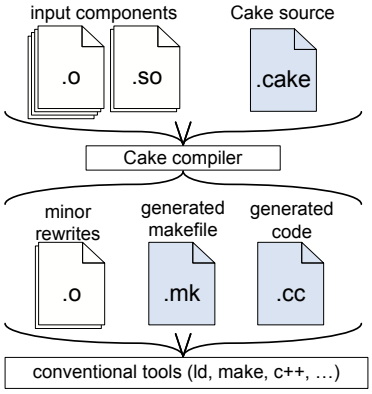

Figure 6. Cake's tool flow

```
exists elf_reloc("foo.o") foo { /* optional info ... */ };
exists elf_reloc("bar.o") bar { /* optional info ... */ };
derive elf_reloc("foobar.o") foobar = link[foo, bar] {
  foo ⟷ bar
  {
    // your correspondences here...
  }
};
```

Figure 7. Skeleton of a simple Cake program

```
1  fopen (fname, "rb")[0]  ⟶  av_open_input_file(
2                               out _, fname);
3
4  values FILE ⟷ AVFormatContext {};
5
6        mpeg2_init() ⟶ { avcodec_init();
7                         av_register_all(); }
8                  ⟵
9        (new mpeg2_dec_s);
```

Figure 8. Some simple Cake correspondence rules

interposing on virtual function dispatch). Supporting *user-defined* styles of object code is planned future work.

3.4 High-level view

Fig. 5 gives a high-level view of the intended result of our video decoding adaptation task: the original components bridged by some Cake-generated adaptation logic. (While our examples show only two components, Cake applies equally well to tasks involving any number.)

Fig. 6 illustrates Cake's place in the toolchain. The Cake compiler inputs a collection of components (in the form of binaries) and some Cake code, and outputs Cake-generated source code, build rules for assembling the output binary (out of this code and the original binaries), and possibly some extended and relinked versions of the original binaries.

Fig. 7 shows the outline of a Cake source file. There are two main top-level constructs: exists and derive. The first of these identifies an existing component—typically a relocatable object file—and optionally adds descriptive information to supplement the debugging information already present. Cake's interface model is based on DWARF 3 [Fre 2005] and its notions of "types" and "subprograms". The availability of debugging information is a huge convenience which we will assume for the purposes of this paper, although Cake does not demand it—all such information can be supplied within the exists block. Certain annotations may also be added (§4.3).

Cake's other essential top-level construct is derive. This describes a new component to be created by assembly and adaptation of existing ones. Derived components are expressed in a simple algebra of built-in functions and operators. The most important of these is link, which applies to a list of component names. All the correspondence rules we have seen would appear inside a block opened by a link keyword, and these account for the vast bulk of any typical Cake program. Since correspondence rules always relate a *pair* of interfaces, rules appear in pairwise blocks, of which there may be many for a given link application (if linking more than two components).

3.5 Syntactic conventions

Arrows in Cake signify correspondence rules, and point in the direction of data flow. In Cake source code, correspondence arrows are rendered using angle brackets and double-hyphens. For example, the bidirectional arrow is <-->. In this paper we typeset them directly as long arrows. Aside from this, Cake's syntax is familiar from other languages, and is mostly C-like. For ease of recognition we typeset all arrow operators specially: -> denotes indirect member selection as in C, and is typeset ↪; meanwhile => denotes functional abstraction as in ML, typeset ⇒; its converse <= (typeset ⇐) binds names to function return values (§4.1).

3.6 Simple correspondences

Corresponding events Lines 1–2 in Figure 8 define an event correspondence, stating that a call to fopen() with second argument "rb" should be translated to a call to

av_open_input_file().[4] The out keyword signifies that the first argument is an "output parameter" into which the logical "return value" of the call will be written; Cake automatically maps this to the return value expected by fopen()—handling of this is discussed in detail later (§4.6). Finally, the [0] qualifier matches only the first fopen() call in the client's execution (since it may want to open other files not for video decoding).

Corresponding values Line 4 says that a AVFormatContext object (on the ffmpeg side) can be created from a FILE object (on the libmpeg2 side) and vice-versa. In this instance, no further rules are specified and no fields are propagated between the two. This is sufficient since the FILE object is completely opaque to the client. If the objects were not opaque, we could add rules inside the braces to describe how their fields relate. In combination with the previous rule, Cake can now generate a wrapper for fopen() which calls av_open_input_file() with appropriate parameters and substitutes the AVFormatContext object with a FILE object on return.

Compound statements and return Lines 6–9 describe initialization of the library state. The pair of ffmpeg initialization calls is given as a compound statement in Cake's "stub language", a simple loop- and recursion-free imperative language. (Although syntactically C-like, this language is completely independent of the components' source languages, since Cake deals only with binaries.) A special *postfix arrow* syntax is provided to describe handling of a return event, here saying that a new object of class mpeg2_dec_s should be allocated on return to mpeg2_init. Again, this object is treated opaquely, so we do not need to describe its fields.

3.7 Remarks on simple Cake usage

We may remark on the usage seen so far.

Dual scoping As befits a language describing relations, Cake has *dual scoping*: different sides of an arrow denote different components, in whose respective scopes names are resolved. The left-hand side of our rules always represent the libmpeg2 client, and the right-hand side represent always the ffmpeg libraries. This means that arrows may point left-to-right or right-to-left, according to which data flow the rule describes. Event correspondences are described using pattern-matching: the arrow-tail side (the "source") represents a pattern that the event matches, perhaps supplying names; these are then bound on the other side (the "sink" side) to the elements they matched in the call. We can bind events to stubs in cases, as in lines 6–7 above, where the function correspondence is not one-to-one.

Rule selection Note that these rules only apply to interactions between our specific pair of components; they say nothing about e.g. how to treat FILE objects passed across other interfaces. In this example the Cake compiler can automatically deduce what value correspondences need to be applied, whereas in a few cases it is necessary to manually instantiate a value correspondence.

Programmer knowledge Like any programming tool, Cake depends on the programmer to understand the semantics of the domain. In writing the above rule, the programmer exploits two facts about the client's usage of the libmpeg2 interface: that it is accompanied by C library calls such as fopen() to do the file I/O, and that the *first* fopen() call opens the video file—signified by the [0] suffix to the pattern.[5] Similarly, the programmer is responsible for writing rules which, in combination, access the ffmpeg interface correctly, e.g. by inserting the av_register_all() call.

3.8 Correspondences for free: name matching

Cake automatically draws *implicit* correspondences between compatible like-named elements in linked interfaces. For example, if one module requires function foo() and another provides it, an event correspondence is automatically drawn between them. This reproduces the behaviour of a conventional linker. Since our current example is an example of *unanticipated composition*, few names match, so the gains from name-matching are modest. However, name-matching is very helpful when applying Cake to *interface evolution*, where many interface elements can be matched without programmer intervention.

Cake extends name-matching to structured values. If two interfaces both define a class bar, then these will be corresponded; if one bar contains fields amplitude, breadth and curvature, and the other breadth, curvature and density, Cake will correspond breadth and curvature, and leave the others uncorresponded. This means that minor mismatches in size or layout of structures are automatically adapted around. For example, the implementation of the C library call fstat() often needs to adapt between kernel- and user-format stat structures, owing only to layout differences and extra fields. Cake could perform this adaptation automatically. In the rare case where a given name-matching is not wanted, it can be overridden by mapping the name to an alternative element (if one exists) or void.

Identifiers often contain meaningful structure. A complementary name-matching feature is the pattern construct, where a single rule can map together sets of similar event names using regular expression matching. The following fictitious example expresses three similar event correspondences in one rule.

```
pattern edit_(cut|copy|paste) (w, sel, ctxt)
  ⟶ clipboard_op_\1 (w, sel, ctxt);
```

[4] Readers familiar with the ffmpeg API may notice that we have used Cake's argument defaulting support to reduce the number of arguments in the second call, for clarity of exposition.

[5] We briefly describe a cleaner approach to this class of rule, based on a more dynamic notion of components defined by generalised *slices* of traces, as future work (§7).

Use of names is sometimes latent rather than explicit. For example, integer fields may implicitly model enumerations or sets; function arguments may also be best understood by their name rather than positionally. Cake supports a names annotation for applying a vocabulary of names (e.g. perhaps from a separate enumeration type, or perhaps given explicitly) to functions or integer fields, in order to induce further name-matching.

4. Advanced features of Cake

We saw in Section 3.2 that simple correspondences are insufficient for realistic tasks like our video decoding example. This section discusses the features of Cake which make it sufficiently powerful to tackle these real-world use-cases.

4.1 Corresponding sequences of events: event context

Often when performing an adaptation, considering each call independently is not enough: the correct action depends on what calls have come before. To this end, Cake event patterns may be prefixed by a *context predicate*: the rule only applies where certain preceding calls have occurred. Automatic management of the state necessary to match such patterns is another way in which Cake saves programmer effort. In our example, we use this facility when the client retrieves an object storing metadata about the video file: we tell Cake that a call to mpeg2_get_info() follows earlier calls to fopen() and mpeg2_init(), whose arguments and return values are significant.

```
// here "..." matches any intervening call sequence
let f = fopen(fname, "rb"), ...,
let dec = mpeg2_init(), ...,
mpeg2_get_info(dec) ⟶// to be continued...
```

Since it may be necessary to refer to values passed or returned during the preceding calls, context predicates can bind names just like event patterns can. The let keyword allows further names to be bound to return values and useful auxiliary values. This does *not* denote assignment (and the same name may not be re-bound within a rule). In patterns like the above which bind names to return values of contextual calls, we can use the shorter varname ⇐ syntax instead of let (see the Appendix for examples).

Resolving ambiguity There is a potential ambiguity in context matching: *which* preceding call is relevant? When a call to mpeg2_get_info() occurs, the libmpeg2 client has not yet associated its decoder with an input stream. However, in ffmpeg the corresponding av_find_stream_info() call requires an input stream as an argument. Somehow, we must match the incoming call with the relevant preceding fopen() call. What if there have been *many* preceding fopen() calls? Cake assumes that related calls occur close together: it matches the *nearest* preceding fopen() (with appropriate arguments). This is expressed using the ellipsis (...) to extend our pattern over unspecified intervening calls. The ellipsis functions much like ".*" within a regular expression, matching any intervening character sequence, but ellipsis matches the *shortest* such sequence rather than the longest. If we had left out the ellipsis, this would match only if the two calls occurred in direct succession (among all calls across this particular interface).

4.2 Generating data-dependent call sequences: stubs

Cake's stub language offers some special features for handling complex data-dependent sequences of calls. These are illustrated by the right-hand side of the the mpeg2_get_info() rule begun in the previous section.

```
/* ... continued */ ⟶ {
    av_find_stream_info(f) // in-place update to f
 ;& let dec...vid_idx = find( // Cake algorithm
      f↪streams, // among the file's streams...
      fn x ⇒ // lambda! find the video stream
        x↪codec↪codec_type == CODEC_TYPE_VIDEO)
 ;& let codec_ctxt = f↪streams[dec...vid_idx]
 ;& let codec = avcodec_find_decoder(
                                codec_ctxt↪codec_id)
 ;& avcodec_open(codec_ctxt, codec)
 ;& codec_ctxt }
```

Error discovery Manually determining the success or failure from every function call can get very tedious. Every expression in the Cake stub language has a "success" or "failure" outcome, logically *separate* from any result value it may yield if successful. Cake determines the success of a function call in a *style-dependent* way (as described in §4.3). The default style assumes that functions returning signed integers are successful iff they return zero, and that pointer-returning functions are successful iff they return non-null. This style is typical of a majority of C APIs. Calls that return neither a signed integer nor a pointer are treated as always succeeding.

Error handling Stubs are generally not complex enough to require try–catch exception handling. Instead, expressions can be joined with short-circuit boolean connectives ;& and ;|, in an idiom similar to that found in Unix shell programming. Unlike the shell, success exists independently of the result value, so the connectives are distinct from the boolean operators && and ||. In the few cases where the style does not detect error status correctly, the programmer can explicitly describe success conditions using the success pseudo-variable and constants void (which yields no value but always succeeds) and fail (which always fails).

Binding Just as let binds names to values in context patterns, it can bind names to values in stubs. These enable data-dependencies between calls. The out keyword also binds a name, and is used when calling functions have output parameters (§4.6).

Associations Sometimes bound names are not enough; a stub must navigate a data structure to find relevant arguments to a call. The dot (.) and short arrow (->, typeset ↪) have the C-like "access member" semantics in Cake. Analogously, the ... syntax is overloaded to denote "access associated": it enables formation and dereferencing of *associations* between objects or values. Associations are the mechanism for many-to-many value correspondences in Cake, and are discussed in Section 4.4.

Lambdas and algorithms Traversing data structures algorithmically is beyond the scope of Cake. However, simple algorithms are often indispensable when performing adaptation. Cake makes a selection of algorithms available in the stub language, here find denoting linear search. Algorithms are defined outside of Cake in an implementation-specific way. Currently, we exploit the fact that Cake's backend generates C++ code: most of the C++ standard library's algorithms may be applied. Cake automatically infers usable iterator definitions using its *style-dependent* notion of lists and arrays (§4.7). Since algorithms sometimes take functions or predicates as arguments, simple functions may be defined as lambdas in the stub language. The expressiveness of this is deliberately constrained: lambdas may not contain other lambdas, and cannot refer to themselves, so cannot introduce recursion in the stub language.

4.3 Practicalities

We have now seen the basics of the Cake language. In this interlude we discuss several practical issues arising in the use of Cake.

Target representation Our chosen binary representation is *relocatable object code*. This means compiled native code, before linking, in any modern containing format such as ELF. Most of our work has applied Cake only to static linking, but its approach applies equally to dynamic linking.

Source languages Cake can compose components deriving from several source languages. In this paper we have targetted only components written in C, since our current implementation lacks understanding of some incidental features found in binaries originating in other languages (such as name-mangling, and various DWARF constructs). Adding such support in most cases is straightforward and is ongoing work. (At run time, some cooperation with garbage collectors is required: see §5.)

Obtaining debugging information Compilers usually require a command-line flag to enable generation of debugging information.[6] Most software builds released to end users do not contain debugging information, but distributors often supply it as an optional extra.[7] We encourage this practice, since there is considerable value in providing debugging information to users (e.g. enabling higher-quality bug reports).

[6] An extended set of flags may be needed to generate the most detailed debugging information available. With `gcc` we have been using `-g3 -fno-eliminate-unused-debug-symbols -fno-eliminate-unused-debug-types`. We also disable inlining, since premature inlining can potentially interfere with Cake. Interprocedural optimisation is best done at link time, when a complete call graph is available.

```
// C declaration
// "foo is a function from int (call it 'a') to int"
int foo(int a);

// Cake description:
// "foo is a function from int (call it 'a') to int"
foo: (a: int) ⇒ int;
// "int is 4 bytes of the 'signed' base type encoding"
int: class_of base signed <4>;
```

Figure 9. Interface description syntax

Interface description As described earlier (§3.4), Cake allows programmers to supplement or replace available debugging information within exists blocks. For this, we devised a simple textual syntax for the relevant subset of DWARF, of which Fig. 9 shows a small fragment.

Annotations The same syntax extends DWARF by accepting certain annotations. For example, attributes out or inout can be made to function arguments, affecting how Cake applies value correspondences to values flowing into and out of a function call (§4.6). Some of these annotations could be useful to debuggers as well as to Cake; we plan to feed these back into the DWARF design process.

Comprehension As with any programming tool, we assume that the programmer understands the interfaces he is coding against. In addition to debugging information, the programmer might use various means to gain this understanding: API documentation, source code, other code exercising the same interfaces, patterns mined from such code [Wasylkowski et al. 2007] or reverse-engineering tools [Balakrishnan et al. 2005]. The latter is especially relevant when Cake is used to compose binaries for which source code is not available. Although these means each have their shortcomings, we consider these as separable problems; in this work we assume that the combination of these techniques is sufficient to gain the necessary understanding.

Styles All components introduced by an exists block are interpreted according to a *style*. Styles are an abstraction mechanism designed to seamlessly support mixing and matching of object code adopting different sets of interface conventions, perhaps originating from multiple *packagings* (e.g. component systems, application plugins, etc.), language implementations and/or coding styles. Styles determine various higher-level interpretations which the Cake compiler applies to object code, including error-handling, treatment of lists, string handling and so on. At present, Cake supports only one style, the "default style", which corresponds to the conventions typically found in components

[7] e.g. in Debian and certain other GNU/Linux distributions

written in C. However, Cake is designed to accommodate multiple user-defined styles in the future.

Instantiate Many clients dynamically load back-end components, such as plug-ins. To use Cake across these interfaces requires a small extra feature. Since the client does not call the back-end *directly*, but through an indirect dispatch table, we provide an instantiate primitive used alongside link in derive expressions. This constructs an instance of a given data structure—usually a dispatch table—and creates a new symbol for each element in the structure. This lifts table entries to first-class symbolic function names which can be used like any other in a link block.

Conveniences The inline construct is similar to exists, but allows a component to be supplied not by reference to an existing file, but by inclusion of a snippet of foreign source code embedded directly in a Cake source file. These snippets are lexed but not parsed by Cake, so any language with compatible lexical structure (up to balanced opening and closing braces) may be used; they are de-lexed and output as source files alongside Cake's other output, and compiled at the same time as Cake-generated code (§5.1).

4.4 Many-to-many value correspondences

In our running example of libmpeg2 and ffmpeg, the structures maintained during decoding by the two libraries contain mostly the same information, but split differently among various objects. In general, while objects or values often do not correspond one-to-one among different interfaces, we can often say that a *group* of objects corresponds to another group. Many-to-many value correspondences describe how to create and update values in one group from (multiple) values in the other. Fig. 10 illustrates this and some other advanced features of value correspondences.

Associations Each many-to-many value correspondence creates *associations* at run time. Each instantiated association is a tuple binding together several objects. Bindings are formed in stubs by applying the let keyword to in combination with the "access associated" connective, written .... These tuples constitute a dynamic relation maintained at run time, analogously with join tables in a relational database. A tuple persists as long as any bound object does.

Initialization versus update Value correspondences may distinguish *initialization* from *update*, as seen in the first rule above. When an object flows across an interface for the first time, Cake may need to instantiate one or more *corresponding objects* (co-objects). Initialization rules use an arrow suffixed with a question mark. When initializing the right-hand side above, p will point to a new AVPacket object. Rules without the question mark are *update rules*. Above, no update rule is needed because the client never updates any state corresponding to AVPacket's fields. Alternatively, sometimes a co-object's fields have no analogous fields in the original object. Cake will initialize these

```
values (dec: mpeg2_dec_s, info: mpeg2_info_s,
 sequence: mpeg2_sequence_s, fbuf: mpeg2_fbuf_s)
  ⟷ (ctxt: AVCodecContext, vid_idx: int,
       p: AVPacket, s: AVStream, codec: AVCodec)
{ // ensure an AVPacket exists on any flow L-to-R
void ⟶?(new AVPacket tie ctxt) p;
// picture dimensions are in sequence and ctxt
sequence ⟷ ctxt {
  // width and height done automatically
  display_width ⟵ width;
  display_height ⟵ height; // here we assume a
  chroma_width ⟵width / 2; // 4:2:2 pixel format,
  chroma_height ⟵ height / 2; };
// info.sequence always points to sequence object
info.sequence (&sequence)⟵? void;
// special conversion required for buffers
fbuf ⟷ frame {
 buf[0] as packed_luma_line[height] ptr
 ⟷ data[0] as padded_line[ctxt.height] ptr;
 buf[1] as packed_chroma_line[chroma_height] ptr
 ⟷ data[1] as padded_line[ctxt.height / 2] ptr;
 buf[2] as packed_chroma_line[chroma_height] ptr
 ⟷ data[2] as padded_line[ctxt.height / 2] ptr;
} };
values packed_luma_line ⟵padded_line {
 void (memcpy(this, that, display_width))⟵ void; };
values packed_chroma_line ⟵padded_line {
 void (memcpy(this, that, chroma_width))⟵ void; };
```

Figure 10. An advanced value correspondence

fields (using the initialization rule), but will subsequently leave them alone (there are no update rules), avoiding repeatedly re-initializing the fields at each traversal of the interface (which might clobber updates made earlier by code on the co-object side). The separation is asymmetric: if there is no separate initialization rule, an update rule will be used, whereas the converse is not true.

Primitive values Cake can usually deduce sensible behaviour for passing primitive values between components, since it inherits from DWARF an understanding of all the common encodings of primitive values like integers, booleans, characters or floating-point data.

Tying The **tie** keyword can be used when allocating objects in Cake, to specify that the allocated object should be deallocated at the same time as the tied-to object. This is a common requirement in Cake, since objects created by adaptation logic are normally tightly dependent on some application-domain object. Tying greatly reduces the need for explicit object freeing in Cake. Tying may be thought of as a generalisation of stack-allocated objects or contained subobjects; in all these cases, one object's lifetime is tied to that of some other allocation. Implementation of tying relies

on the Cake runtime's ability to interpose on object deallocation, which is also used heavily by the Cake runtime internally (§5.4.2).

Internal reference The unusual-looking rule describing info.sequence is used to describe the pointer structures *within* a group of objects. When creating an mpeg2_info_s structure, Cake needs to know that its sequence field should point to the related mpeg2_sequence_s structure. Since this does not depend on any value from the right-hand side, void appears (to denote "no value"), but the syntax is otherwise identical to any other correspondence.

Applying functions A bracketed stub-language expression on one side of an arrow may be used to apply a function to the outcome of the source side, for cases where some computation is required in order to acquire the correct sink-side representation. (Consider a sink-side field storing an index into a table, while the source provides only a pointer to the corresponding element—a search through the table is required.) Such functions may be applied either *before* or *after* the conversions implied by other rules, according to which side of the arrow the expression appears on.

This and that The this and that keywords are pointers to the local and (respectively) other-side representations of the value (i.e. before and after applications of the value correspondence). These pointers can be useful when applying functions as values traverse the interface. In the example above, both interfaces describe some buffers containing the decoded data, but with a subtlety: the layout of the buffers is not quite identical. In ffmpeg each line is padded, whereas in libmpeg2 there is no padding. The code above uses the as keyword, which implicitly defines a new named class of value (equivalent to a DWARF type) and treats the value as if it were of that class. We can then supply special value correspondences for that class—here these override the default handling of uint8_t arrays, using memcpy() on this and that to move the data in a padding-sensitive fashion.

4.5 Adapting constants

Sometimes, correspondences occur at the level of individual values. Consider explicitly mapping the individual elements of two enumeration types. Cake provides a table construct for this purpose, syntactically much like a value correspondence but mapping constants or literals rather than fields.

Components often statically embed meaningful static data (e.g. configuration file paths) which need to be adapted. We support unilateral adaptation of these by rewrite rules in exists blocks. Strings can be matched by regular expressions, and updated in the object file by appending new static data and rebinding existing references. This is the only form of binary rewriting done by Cake.

4.6 Input and output parameters

Pointers are used to perform certain forms of parameter-passing. Cake's default style assumes (unless overridden)

```
strncpy:
 (dest: out char[len], buf: char ptr, len: size_t)
  ⇒ char[len] ptr;

strndup:
 (buf: char[] ptr, max_len: size_t)
  ⇒ char[] caller_free(free) ptr;
```

Figure 11. Enabling allocation adaptations on strncpy() and strndup()

that singly-indirected arguments typically denote "in-place update"—a value passes *out* of the call as well as *in*. Cake will apply the appropriate value correspondences on both entry and exit. These semantics ensure that balanced operations can be reliably expressed (e.g. to insert locking and unlocking as an object traverses an interface).

A singly-indirected argument might also denote an *output parameter* (typically passing an uninitialised location in the caller's stack frame). Since the in direction may perform a (meaningless) conversion on contents of the stack location, the argument can be annotated as out. Similarly, for objects no longer valid *after* a call (e.g. if deallocated during the call), we can annotate the pointer as an in ptr argument, preventing any output value correspondence from running.

Commonly, output values flow into caller-allocated memory. However, some interfaces return callee-allocated results. Given simple annotations, Cake can automatically adapt between mismatches in these caller-versus-callee allocation semantics. Consider the C library functions strncpy() and strndup() (shown in Fig. 11). In the first, a caller supplies its own buffer for output data to be placed in. The second instead returns a pointer to a new buffer, which the caller must free when finished.

Thanks to the caller_free annotation, Cake will adapt a call to the first function so that it instead calls the second, by post-copying the callee-provided buffer into the caller-provided buffer and then freeing the former. Cake can also adapt a call to the second function so that it calls the first, by pre-allocating a buffer and returning it.

4.7 Arrays and lists

Pointers may also point to arrays; as seen in Fig. 11 with char[], array syntax can be used to name a local field which holds the length of the array. Cake usually detects the size of arrays at run time and applies appropriate conversions (§5.3).

Cake also has a style-dependent notion of *iterables*; in the default style this includes arrays (either statically-sized, length-affixed, or explicitly terminated as with null-terminated strings) and linked lists. These allow algorithms (find seen in Section 4.2) to be applied uniformly to any style-defined iterable.

4.8 Function pointers

Functions are just another kind of object. Although their internal structure is opaque to Cake, we already have a mechanism for describing correspondences between functions, namely, event correspondences. Passing a function pointer is equivalent to giving the recipient a capability to raise events across the interface between a pair of components. Therefore, Cake will handle flow of function pointers appropriately provided there is some event correspondence defined across that interface whose sink expression is a simple invocation of the passed function (rather than, say, a multi-expression stub). Consider the following example.

```
1  client ⟷ library
2  { register_callback(f, arg) ⟶ add_handler(f, arg);
3    notify_user_cb(message, aux) ⟵ _(message, aux); }
```

Here the developer describes how a function pointer may be passed from client to library by register_callback(). Line 2 adapts a mismatch in the callback registration interface: the client requires a call named register_callback whereas the library provides only a similar function called add_handler. For simplicity our rules assume that the callback interface itself (e.g. the signature of functions passed as the f parameter) is well-matched between the two interfaces, but if additional adaptations are required on the function, they can be added as a lambda expression (e.g. wrapped around f in add_handler above).

The event correspondence in line 3 is unusual because it does not specify a name for the called function, but simply uses the "_" syntax, meaning "some call" . This is because the because the call-site in the library is an indirect call, so does not statically name the function it is calling. Without line 3, or some other rule calling notify_user_cb from library, the Cake compiler would not generate code to interpose on the callback—e.g. to apply the value correspondences appropriate for the two components. The presence of this rule enables function pointers (such as notify_user_cb) to be correctly adapted as they are passed to the library, by substituting a pointer to a Cake-generated wrapper which applies the relevant correspondences.

4.9 Completing the example

One hurdle remains in our running example, which is to match up the decoder loops of the two interfaces use patterns. This requires no new Cake language features, so for space reasons we have left it to the Appendix, which contains the example's Cake code in full.

5. Implementation

We discuss the compiler implementation briefly, and the Cake runtime in detail.

5.1 Compiler back-end

Cake models a program as a set of communicating object files—or more properly, groups of objects files, which we call components. Communication occurs along the control path of the program; an "event" between two components' interfaces occurs when control flows out of one component and into another. Cake is implemented by interposing on these events: Cake-generated code runs when events occur. (The defining characteristic of a component is that it is internally well-matched with itself—no interposition is necessary on communication *within* the component.)

The current implementation of Cake assumes that inter-component data flow occurs only through function calls, *or* through shared objects whose sharing was established at run time through function calls (e.g. by passing a pointer in an earlier function call). This allows Cake to intervene at the point where sharing is established. We discuss this further in Section 5.4. The assumption might be violated by statically allocated shared variables, since sharing is established at link time. In practice, globals shared among components are rare. Where they do exist, this interface is usually that of a standard library (e.g. the C library's errno) rather than one suffering mismatch.

Our current Cake compiler's back-end uses a specially created dwarfhpp tool to generate C++ headers which reproduce the ABIs described in DWARF information, using compiler-specific attributes to match alignment and encoding where necessary. The Cake compiler outputs wrappers in the form of the C++ code consuming these headers, and a POSIX makefile. Cake's algorithms and lambdas map conveniently onto those provided by C++. Wrappers are interposed using the linker's –wrap option, and we have prototyped a similar mechanism using the dynamic linker's LD_PRELOAD setting for the dynamic linking case. To perform string rewriting and occasional other symbol rebinding, Cake's back-end uses a specially modified version of GNU objcopy [8].

5.2 Compiler status

At the time of writing, our Cake implementation is not complete, but is progressing fast. Code generation for is implemented for the simpler kinds of correspondences and is a work in progress for the remainder. The compiler back-end machinery, including dwarfhpp and the modified objcopy, is complete. The Cake runtime is also very usable and has been used successfully in some earlier case studies involving script-generated wrapper code.[9]

5.3 Dynamic binding

When control passes from one component to another, Cake behaves as if the program's entire object graph is carried

[8] http://ww.gnu.org/software/binutils/

[9] We encourage the reader to check for software releases at the Cake web page, http://www.cl.cam.ac.uk/%7esrk31/cake/.

over and transformed according to the set of value correspondences. Its *actual* behaviour is subtler and less inefficient (§5.4.2).

Objects in native code are not self-describing at run time, and the debugging information which describes them, much like static types, is inherently imprecise. If debug information says that a function returns a pointer to a Widget, and Window is a subclass of Widget, the function might actually return a Window pointer. Suppose we write some Cake rules to adapt between two different implementations of a similar windowing toolkit.

```
widgets_A ⟷ widgets_B
{ // an event correspondence
  find_widget(descr) ⟶ get_matching_widget(descr);
  values Widget ⟷ Widget
  { /* ... */ };
  values Window ⟷ Window
  { /* ... */ };
}
```

Cake has dynamic matching semantics. If the pointed-to object "is a" Window, then Window's rules must apply. We achieve this by defining an "is a" relationship between DWARF types. In turn, this means assuming a certain "well-behavedness" of the target code: DWARF information may be imprecise, but not wilfully misleading. Arrays compound this difficulty: does a pointer point to one object, or to (or into) a block containing several? If Cake decides incorrectly, it will not apply the correct conversions. In summary, the Cake runtime must be able to decide two questions about objects.

- Given a pointer to an object, what *byte-scale reinterpretations* might a component reasonably make, to reveal a pointer to a larger object?
- Given a pointer to an object, what *block-scale adjustments* might a component make, to navigate among objects in the same array?

The two are not independent: to apply pointer arithmetic, a component must know the element size, so we assume that a component may *not* do *both* byte- and block-scale reinterpretations (unless the Cake programmer provides a precise type by annotation).[10]

Cake's rules about allowable byte-scale reinterpretations define what we call *admissible reinterpretations*. They are designed to separate out common-case "well-behaved" pointer adjustments from uncommon cases requiring annotation. We call these uncommon cases "abstraction violating" after prior work [Neamtiu et al. 2006] which also provides evidence that they are suitably rare.

[10] To do both would be to access an array through a pointer whose static type did not reflect the true element size. In our experience this occurs only when a function receives an array as a void*, but accesses the array by strengthening that type. Cake's annotations handle this case conveniently.

For a pointer whose referent is statically typed with type τ, admissible reinterpretations are as follows.

- If τ is primitive, no reinterpretations are admissible.
- If τ is structured, reinterpretations to any *zero-offset containing type* are admissible. A zero-offset containing type is one which contains a subobject of type τ at offset zero. We allow this to support the idiom often found in C-language object systems[11] which simulate inheritance by zero-offset containment.
- If τ is structured, reinterpretations to any DWARF-recorded inheriting type are admissible. This allows for downcasts using DWARF's special inheritance tag (which supports single or multiple inheritance).

To discover the *most precise* DWARF type for a given pointer, we use knowledge of address space layout to deduce whether the object is in heap, stack or static storage. For the latter two cases, a precise type is found in debugging information (for the allocating stack frame or static variable definition). This also reveals whether the object is part of an array.[12] In the heap-allocated case, we exploit our assumption that the recipient might do *either* byte-scale *or* block-scale adjustment (but not both) to derive a best-effort solution using linear programming, as follows.

1. Discover the size and start address of the object's heap block in bytes, using implementation-specific knowledge of the heap. (This requires that custom allocators be instrumented to collaborate with the Cake runtime.) If the start address and size match the pointer and its static type, we have an answer.
2. Else test whether the heap block is an array (block-scale adjustment): if its size is a multiple of this static type's size, *and* the pointer's offset into the heap block is a nonnegative integer multiple of that size, assume we have a pointer into an array occupying the entire heap block.
3. Else test whether the heap block is a containing object (byte-scale adjustment): compute what (admissible) *containing types* would have sizes matching the heap block size *and* consistent offset. If there is a unique match, we are finished: we assume that the adjusted static type precisely describes the object.

If this fails, we issue a run-time warning and proceed with an imprecise type. This is often not a problem, depending on how the receiving component interprets the pointer. We consider the unknowability of precise run-time types for heap-allocated objects as a weakness in language runtime design, which hinders not only Cake but also debuggers, garbage collectors and other dynamic analyses. The omission is often deliberate—for example, the C language defi-

[11] A popular example is GObject, http://www.gtk.org/

[12] Note that we are now requiring that debugging information, or equivalent annotation, be available *at run time*.

nition explicitly disclaims the existence of a definitive interpretation for any memory location. However, in reality, programmers nearly always *do* have such definitive interpretations in mind. Compile-time analyses could generate precise heap metadata in most cases (e.g. by examining data flow out of malloc()), and issue warnings in others.

A similar problem occurs with unions: which arm of the union is the currently valid representation? Nontrivial use of unions is sufficiently rare that we have left this to future work. The best treatment is to re-encode union types as Pascal-style variant types: these are supported by DWARF, and the re-encoding can be expressed as annotations in a Cake exists block.

5.4 Adapting objects

We consider objects to be structured values with *two* key additional properties: identity and lifetime.

5.4.1 Object identity

Cake understands objects' addresses in memory as their identities. At run time, it maintains a table called the *co-object relation* which maps related identities to each other. As pointers pass across an interface, Cake substitutes pointers to appropriate *co-objects*. Usually, for a given tuple in the co-object relation, exactly *one* object was allocated by user code; the others were allocated by Cake when a pointer to the first object, or some subseqent co-object, was passed. *Associations* (§4.4) are implemented by mapping each object to an Cake-generated *umbrella object* which contains pointers to other objects in the association. At present this constrains an object to be participating in at most one association at a time, although we intend to relax this to one per correspondence rule, by giving each association rule a run-time tag.

5.4.2 Object lifetime

When applying value correspondences to produce transformed versions of objects, Cake must allocate memory. This memory has a lifetime *tied* to the user-managed objects that caused its allocation. Implementing tying requires interposing on object deallocation. In the case of heap deallocation (with free() or other heap-specific mechanism) this is straightforward. For stack-allocated objects, Cake must interpose on cleanup of the allocating stack frame. This is implemented by replacing the on-stack return address for the allocating frame with the address of a handler. This handler uses the stack pointer to identify which frame is returning, deallocates any tied objects, and jumps back to the intended return address.

5.4.3 Sharing objects

As described so far, each component appears to have its own heap, completely separate from other components'. In fact, Cake allows sharing of objects between components, subject to the invariant that each component can only reach objects whose representation it understands. (We define this more precisely below.) The effect is a *partially split heap*—some objects are shared, and others are replicated (perhaps in alternative representations).

Enforcing this invariant is nontrivial because of the transitivity of reachability. We start by partitioning the (infinite) set of run-time objects into equivalence classes based on their "most precise DWARF type" (§5.3). Our question then becomes whether Cake can allow two components to share an object of a given class. We only have space to outline the intuition behind our algorithm here. Firstly, consider the DWARF types of all objects which are *related* between each pair of interfaces. We call this the *master type relation* for that pair, and it is enumerated by the set of of value correspondences established between the two components (including those made by name-matching). Next, we define a binary relation *representation compatibility* on DWARF types, recursively as follows.

- For a structured type: if the two structures define identical sets of field names at identical offsets, and for each like-named field the field's type is representation-compatible, then the structures are representation compatible.
- For a pointer type: all pointers are representation compatible. We account for reachability in a separate step (below).
- For a primitive type, the types are representation compatible if and only if size and encoding match exactly.[13]

The "possibly shareable" set is those pairs in the master type relation that are representation compatible. Not all of these are actually shareable, because they might contain pointers to objects which are not shareable. We generate the "definitely shareable" from the "possibly shareable" set by removing (until a fixed point) pairs where, given a pointer to some shared object, both components could reach some piece of memory about which their expectations are not representation compatible. We do this by considering the *type reachability graph* as the connected digraph (V, E) where E includes (v_1, v_2) iff a pointer to type v_1 can yield a pointer to type v_2 by *either* member selection *or* an admissible reinterpretation (§5.3). We must label each edge to identify which member was selected or what interpretation was applied, then remove any (α, β) if there exist some *non-shareable* α' and β' reachable respectively from α and β by analogous paths in each's type reachability graph.

We conclude this discussion with a few notes.

Opaque and ignored pointers A technique complementing this algorithm is to obtain more precise information about the interpretations *each component* makes of its objects. If a component always ignores some field in an object, or treats a pointer opaquely, this can enable more shar-

[13] A subtlety here is enumerations, bitfields and other encodings layered onto primitive types. We rely on programmer annotation to interpret these, for example using the names construct (Section 3.8).

ing. We are working on support for this using programmer-supplied opaque and ignored annotations; future work could infer these by analysis.

Update propagation and multithreading The "partially split heap" is compatible with multithreaded programming, but our current propagation policy is not. Specifically, we currently use a policy of propagating updates between *all* replicas whenever control passes between components; this is correct in the single-threaded case, although slow (because of potentially high update volumes at each interface crossing). To reduce the update volume, points-to analysis could produce a tighter bound on which objects' updates may be needed during a given call. Our policy also risks deadlock in multithreaded programs where more than one component contains active threads at a given instant, since depending on the program's control flow, updates may never be propagated. A periodic background sync thread could ensure liveness, but since this might activate mid-update, ensuring safety is a problem: it could most likely be solved like the analogous problem in dynamic software update systems, using "quiescent update points" [Neamtiu et al. 2006] and programmmer-annotated "propagation points" [Neamtiu and Hicks 2009]. A final problem in the multithreaded case is conflicting updates to separate replicas of (logically) shared state. To solve this, shared-writeable objects could be managed using an alternative replication-free approach, using memory protection techniques to trap updates.

6. Evaluation

Cake's major advance is as a convenient, powerful adaptation tool which can be applied to real-world tasks. We therefore evaluate it by identifying a series of example tasks which have *already* been performed using conventional approaches, and comparing this code to the equivalent Cake code. We discuss each task briefly and report aggregate measurements for both versions (lines-of-code counts, token and statement counts). Since we currently lack a complete implementation of Cake, reimplementing existing adapters in this way is useful, because we can nevertheless gain reasonable assurance that our Cake code is complete by checking that all of the original logic is accounted for in the Cake version.

Measurements Although we use count-based measurements, we appreciate their shortcomings. Cake's lower counts certainly originate partly in improved abstraction, but perhaps also to incidental factors such as a reduction in boilerplate code. We have partially remedied this by providing "adjusted" counts for C code, made after erasing common C boilerplate (specifically, variable declarations and function prototypes), but this is an ad-hoc adjustment which still does not account for certain other areas where Cake's syntax may be more concise (e.g. error-path control flow). The "remaining" column in our tables refers to C code that could not be reimplemented in Cake and was left as-is (to be linked alongside the Cake-generated code).

Limitations Even if Cake did little to simplify code (which is far from true), there are inherent benefits in Cake's black-box, binary approach which are not substantially evaluated here. Our goals with Cake are not simply to provide a marginally better way of coding adapters, but rather to enable a shift in development practices towards integration-based approaches and away from reimplementation and invasive editing. Clearly this cannot be achieved or evaluated in small-scale studies.

6.1 Bridging related components: libp2k

Filesystems are a ubiquitous abstraction: filesystem-like interfaces are implemented deep within operating system kernels, but also in graphical desktop environments, in web servers and elsewhere. The programming interfaces behind which filesystems are implemented are invariably *abstractly similar* yet often *concretely different*, and conventionally coded adapters exist between some of them. We took the libp2k adapter [Kantee 2009] from NetBSD, which adapts between NetBSD's native user-space filesystem implementation (puffs) and a special environment for running unmodified kernel code, including filesystems, in user-space (rump), and reimplemented it using Cake.

Figure 12 shows a large portion of the Cake code for this task. We were fortunate to have a one-to-one correspondence between most calls in the two interfaces, with well-matched naming conventions; this is captured neatly in two pattern rules (labelled "hunk 1").

There are some simple correspondences between objects in the two interfaces (hunk 2). Some rump library calls leave their vnode target unlocked, so we need not apply RUMP_VOP_UNLOCK() in those cases (hunk 3). These calls are exactly those which may modify the filesystem's directory structure; such calls also require the reference count of any *modified* vnode to be pre-incremented to avoid premature reclamation, as captured by the vnode_bump rules.

Some rump functions return output values through parameters (hunk 4a). The puffs interface requires these to be passed through an opaque object, puffs_newinfo, populated using setter functions. We can express this firstly by describing which rump calls' arguments are outputs, and secondly by providing value correspondences between puffs_newinfo and the relevant rump structures.

Code in librump originated in the kernel, where client reading and writing of file data requires address-space traversal. The four relevant calls use a special interface called uio for passing this data. To us, this is just a new way of packaging parameters for input and/or output, and is handled by a few more correspondences.

The rules shown generate complete implementations of all but six of the 28 p2k wrappers. The omissions are explained by special error-handling requirements, one-to-many

```
// hunk 1: basic event correspondence patterns
pattern puffs_fs_(.*) { names (mount: _) }
     ⟷ rump_vfs_\1 { names (mount: _) };
pattern puffs_node_(.*) { names (mount: _, cookie: _ as
                                  vnode_unlocked ptr) }
     ⟷ RUMP_VOP_\U\1\E { names (cookie: _) };

// hunk 2: basic value correspondences
values puffs_usermount (puffs_getspecific(this)) ⟶ mount;
values puffs_cred (cred_create(this)) ⟶ kauth_cred;
values puffs_cred ⟵(cred_destroy(this)) kauth_cred;

// hunk 3: more value corresps incl. special unlocked-return
values vnode_unlocked ⟶
 ({RUMP_VOP_LOCK(that, LK_EXCLUSIVE); that}) vnode;
values vnode_unlocked ⟵(RUMP_VOP_UNLOCK(that, 0)) vnode;
values puffs_cn (makecn(this)) ⟶ component_name;
values puffs_cn ⟵(freecn(this, 0)) component_name;
values vnode_bump ⟶
 ({RUMP_VOP_LOCK(that, LK_EXCLUSIVE);
                rump_vp_incref(that); that}) vnode;
values vnode_bump ⟵vnode; // unlock not required
puffs_node_create(mount, vn as vnode_bump, ni, cn, vap)
  ⟶ RUMP_VOP_CREATE(vn, ni, cn, vap);
puffs_node_mknod(mount, vn as vnode_bump, ni, cn, vap)
  ⟶ RUMP_VOP_MKNOD(vn, ni, cn, vap);
// ... similar for remove, link, rename, ...

// hunk 4a: how to output parameters by "newinfo"
values puffs_newinfo ({puffs_newinfo_setcookie(this, that); this})
  ⟵(RUMP_VOP_UNLOCK(this, 0)) vnode;
// Some calls return a fuller set of newinfo
values puffs_full_newinfo ({puffs_newinfo_setcookie(this, that);
 puffs_newinfo_setvtype(this, vtype);
 puffs_newinfo_setsize(this, vsize);
 puffs_newinfo_setrdev(this, rdev); this}) ⟵
  ({let (vtype, vsize, rdev) = rump_getvninfo(this); this}) vnode
// hunk 4b: tell Cake which calls need "full" newinfo...
exists elf_archive("puffs.a") puffs; // this hunk would appear at
derive elf_archive puffs_inst = // ...the top of the .cake file
  instantiate(puffs, puffs_ops, pops, "puffs");
puffs_inst { declare {
    puffs_fs_fhtonode : (_, _, _,
     out puffs_newinfo as puffs_full_newinfo) ⇒ _;
    puffs_node_lookup : (_, _,
     out puffs_newinfo as puffs_full_newinfo, _) ⇒ _; } };

// hunk 5: shared locking
values vnode_lkshared
  ⟶({RUMP_VOP_LOCK(that, LK_SHARED); that}) vnode;
values vnode_lkshared ⟵({
 RUMP_VOP_UNLOCK(that, 0); that}) vnode;

// hunk 6: input/output by uio
values uio_outbuf (buf: uint8_t[] ptr, resid: size_t ptr,
 off: const off_t) ⟶(rump_uio_setup(that↪buf,
 *that↪resid, that↪offset, RUMPUIO_READ)) uio;
values uio_outresult ⟵(rump_uio_free(this)) uio;
values uio_outres_len_off ⟵({rump_uio_getresid(that↪resid);
     rump_uio_getoff(&that↪readoff);
     rump_uio_free(this)}) uio;
puffs_node_read(mount, vn as vnode_lkshared,
 uio as uio_outbuf(buf, resid, offset),
    _, resid out_as uio_outresult, cr, ioflag)
⟶ RUMP_VOP_READ(vn, uio, ioflag, cr);
// similar: readlink, readdir, "uio_inbuf" and write
```

Figure 12. Selected rules from the p2k study

	C	adjusted	Cake	remaining C
LoC (nb nc)	605	523	133	54
tokens	3469	3137	1131	347
semicolons	358	277	69	33

Table 1. Comparing p2k implementations in Cake and C.

function mappings, and function correspondences which do not follow the naming convention. They were easily handled by a few more event correspondences (not shown).

In summary, Cake can express the p2k component in a fraction of the code size, and in a way which localises each concern of the two interfaces' syntactic and semantic differences far more clearly than the existing p2k code. For example, treatment of unlocking and reference counting is handled by discrete and localised rules, rather than being scattered throughout the code. The only logic required which Cake couldn't adequately express was about 40 lines of C code in p2k.c which load the filesystem (the p2k_run_fs() function). This loader is necessary because puffs only calls into p2k indirectly, through a table of function pointers passed during initialization. We instantiate this table using Cake's instantiate helper (§4.3).

Table 1 shows the aggregate comparison of Cake's p2k with the original implementation.

6.2 Migration between support libraries: ephy–webkit

Another area of continuing evolution is in web browsers. The Epiphany web browser[14] migrated during 2007–08 from a Mozilla-based HTML display widget to supporting aditionally a Webkit-based one. We compare Epiphany's internal WebKitEmbed adaptation layer with a Cake implementation.

Since the developers of Epiphany chose to strip out the adaptation layer around July 2008, after Webkit migration was completed, to target Webkit APIs directly, we used Subversion revision 8300 (28 June 2008) and isolated the adaptation logic in class WebKitEmbed for Cake reimplementation. (Although there is no relevant discussion in the changelogs or mailing list archives, clearly the developers anticipated no future need to change the target API; this strikes us as optimistic.) For simplicity, we left as-is some additional adaptation code handling cookie management, password management and certain other functionality, since this contained only no-op implementations in our chosen revision. Similarly, we retained the utility classes WebKitEmbedPrefs and WebKitEmbedHistoryItem for use by our adaptation logic; these could be implemented in Cake, but owing to their small size, their C code is dominated by boilerplate, so would not give a useful measurements.

Epiphany uses subclassing (using the GObject library) to connect an Embed object with a Webkit instance: the subclass's fields point to Webkit resources. In Cake we use an

[14] http://www.gnome.org/projects/epiphany/

association: the Embed object is associated with the relevant Webkit objects.

```
values EphyEmbed ⟷(web_view: WebKitWebView,
                    scrolled_window: GtkScrolledWindow,
                    load_state: WebKitEmbedLoadState,
                    loading_uri: char []);
```

Most of the calls between the two interfaces map very directly. Some are left unimplemented by Epiphany; these are mapped to empty stubs in Cake.

```
ephy_load(embed, url as raw_url, flags, preview_embed)
  ⟶ { let embed...loading_url = url;
      webkit_web_view_open(embed...web_view, url); };

ephy_stop_load(embed) ⟶
  webkit_web_view_stop_loading(embed...web_view);
ephy_can_go_back(embed) ⟶
  webkit_web_view_can_go_back(embed...web_view);
ephy_can_go_forward(embed) ⟶
  webkit_web_view_can_go_forward(embed...web_view);
ephy_can_go_up(embed) ⟶{ false };
```

The two components exchange history item objects. Value correspondences are provided. Note that in both cases these are passed as GList objects, but with different payload types. Our object-sharing analysis (§5.4.3) correctly catches this: the list nodes are not shared. We rely on explicit specialisation of the void pointers in each GList node. Without this, the pointed-to objects would not be explored by the Cake runtime.

```
// in Epiphany "exists"
ephy_get_forward_history: (_) ⇒GList_of_EphyHistoryItems;
// in Webkit "exists"
webkit_web_back_forward_list_get_forward_list_with_limits:
      (_) ⇒ GList_of_WebKitWebHistoryItems;

// in "link"
ephy_get_forward_history(embed)
  ⟶ {
  let full_list =
webkit_web_view_get_back_forward_list(embed...web_view);
  let copied_sublist =
webkit_web_back_forward_list_get_forward_list_with_limits(
      full_list , WEBKIT_BACK_FORWARD_LIMIT);
  copied_sublist };
```

Epiphany provides code to manually walks and convert the two history lists. Our code simply treats the list as an object graph and applies the relevant value correspondences. Pattern-matching on event correspondences also simplifies the load and manager_do_command functions. Finally, a small benefit in the Cake implementation is a relative lack of boilerplate: whereas Epiphany's use of the GObject library necessitates somewhat verbose C code to perform downcasts and populate a dispatch table, by contrast Cake can succinctly instantiate the table using instantiate. Since associations are formed dynamically and navigated using run-time metadata, downcasts are unnecessary.

This case study proves a fair demonstration of Cake. However, since the data passed back and forth between the browser and its back-end are relatively simple, Cake's powerful rule-based value conversions do not pay off as heavily as in p2k. Table 2 shows the aggregate comparison of the original implementation and Cake's.

	C	adjusted	Cake	remaining C
LoC (nb nc)	525	513	161	0
tokens	2529	2455	784	0
semicolons	175	163	70	0

Table 2. Comparison of ephy–webkit in Cake and C.

6.3 Evolving interfaces in distributed systems: XCL

Codebases in long-lived distributed systems accumulate complexity over time. Occasionally developers choose to redesign the client interfaces to shed this complexity and better serve current needs. Such an initiative began in the X Window System around 2003, when a new client library XCB was proposed to replace Xlib. For clients of Xlib, an adaptation layer called XCL [Sharp and Massey 2002] was devised. We took a small but representative subset of the XCL source code (around 600 raw lines out of 6000) and reimplemented it using Cake.

Since XCB is designed to be more minimal than Xlib, there is a small abstraction gap between the two. As a result, some utility code from XCL whose purpose was to bridge that gap was retained unmodified for use with our Cake implementation. Meanwhile, many data structures are shared verbatim between Xlib and XCL, so there was only limited opportunity to exploit the expressiveness of value correspondences.

This study exposed a flaw with the current Cake language: it has no means to factor out cross-rule commonality which cannot be captured using value correspondences. In XCL there is some such commonality. For example, several Xlib calls for setting window properties map to the XCBChangeProperty call, which takes many arguments. In XCL, there is an XSetProperty function which abstracts away most of these arguments, and series of other Xlib calls are implemented using this function. In Cake we were forced to implement each as a verbose call to XCBChangeProperty instead, making the Cake version longer than the C version.

```
// longhand in Cake, repeating the XCBChangeProperty call
XSetWMName(dpy, w, tp) ⟶XCBChangeProperty(
      dpy, PropModeReplace,
      w, XA_WM_NAME, tp↪encoding,
      tp↪format, tp↪nitems, tp↪value);


// shorthand in C, using XSetTextProperty convenience
void XSetWMName(Display *dpy, Window w, XTextProperty *tp)
{ XSetTextProperty(dpy, w, tp, XA_WM_NAME); }
```

Table 3 shows the aggregate comparison of the original implementation and Cake's. We were disappointed not to make bigger savings in this study. The abstraction gap contributed some additional complexity to the Cake code, as did

	C	adjusted	Cake	remaining C
LoC (nb nc)	380	315	189	42
tokens	2581	2328	1543	232
semicolons	187	148	107	19

Table 3. Comparison of an XCL subset in Cake and C.

the asynchronous style of dispatch in the XCB interface, and the fact that Xlib's return conventions, which return 1 on success, do not match Cake's default style of error reporting (§4.2). We hope that support for styles in Cake will be able to abstract these more cleanly in the future. Had we had the resources to implement the whole of XCL in Cake, we would expect better figures, since greater commonality would be extracted by value correspondences.

7. Discussion and future work

Performance Achievable performance using Cake depends greatly on the "cut" of the interfaces being composed. We have several reasons to believe that Cake's generated code can be efficient in many cases: it is often remarkably similar to hand-written code (particularly the p2k study), and link-time optimisations can be applied after Cake has done its work. The relatively slow uptake of link-time optimisation suggests that cross-library calls are rarely performance-critical (cf. intra-library calls). Finally, there is huge scope for adding further annotations and analysis to allow generation of faster code.

Applicability Cake's range of applicability can only be discovered in the longer-term, but its underlying model is highly general and certainly not limited to procedural interaction. Cake's design might apply particularly well to distributed systems where storage is naturally replicated more than it is shared.

White-box complement Cake only tackles *well-abstracted* tasks (§3). Binary instrumentation systems, such as Pin [Luk et al. 2005] could make an extremely useful complement to Cake for turning ill- into well-abstracted tasks, but we have yet to investigate this.

Dynamic component structure Cake currently identifies "components" by interfaces visible statically in object code. Often, however, the same static set of functions and datatypes can realise logically quite different components at run time. For example, two FILE objects might each constitute a logical component to which different adaptation rules should be applied. A refined notion of component interfaces as "slices" of a trace, identified by patterns much like event sequences (§4.1), could support such use-cases.

Binaries and styles The programmer must understand two versions of their interface: source-level and binary-level. With C code (the "default style" target) these two views are usually very similar, but can be obscured by use of the preprocessor (e.g. to redirect function calls). Styles become more indispensable when targetting components written in other languages. For example, comfortable support for C++ and Java requires styles to interpret name-mangling conventions, virtual function dispatch and exception handling. Our immediate future work is to tackle these and related issues.

Scale Our evaluation case studies are relatively small. However, we would expect interface size or "surface area" to grow sublinearly with both component size and program size ("volume"). A deeper study of this is warranted.

Bidirectionality Currently in Cake, only the simplest correspondences may easily be made bidirectional. In future work we hope to unify stubs and patterns somewhat, so that more rules can be naturally bidirectional. For instance, a stub which does `a(); b()` can be treated as a pattern which matches the sequence `a(), b()` in the reverse direction. Stubs which restrict themselves to reversible programming constructs, much like *lenses* [Bohannon et al. 2008], could be interchangeably rendered as patterns in this way.

Automation Cake's correspondences are effectively a somewhat strengthened *specification* such as might be fed to a converter synthesis algorithm [Passerone et al. 2002]. Still missing is a description of the *protocols* of the input components, so that the trickier aspects of control structure can be inferred automatically. Recent work on object usage pattern mining [Wasylkowski et al. 2007] extracts exactly this information; this could be a basis for greater automation of Cake coding.

8. Related work

Cake is primarily an adaptation tool, and combines many of the techniques in foundational work on procedural adaptation [Purtilo and Atlee 1991] and protocol adaptation [Yellin and Strom 1997, Passerone et al. 2002, Bracciali et al. 2005]. Jigsaw [Bracha et al. 1993] is an early system proposing limited adaptation and composition abstractions for binaries. None of this work has Cake's support for complex object structures (§5), nor many-to-many correspondences among values (§4.4), nor Cake's level of expressiveness in data-dependent function correspondences. More generally, none presents a complete and practical tool design nor experimental evaluation on realistic use-cases.

Cake's interface correspondence rules are similar to *composition rules* found in subject-oriented programming [Harrison and Ossher 1993], although the latter is neither a black-box approach (since rules may range over all source artifacts) nor specialised for adaptation tasks. Cake's notion of value correspondences is somewhat similar to "typemaps" in Swig [Beazley 1996]. Swig targets a strictly smaller problem (interoperation between C/C++ and scripting languages) than Cake, and has a clear directional bias (the script interface is *generated from* the C one) which constraints its order of application. Cake's treatment of pointers, being able to

translate entire object graphs at a time, is far more expressive than Swig's.

Recent work has furthered adaptation as a language feature in C++ code [Järvi et al. 2007], as a compatibility technique in cooperation with refactoring tools [Dig et al. 2008] and as a source-to-source translation for Java code [Nita and Notkin 2010]. The latter work, Nita and Notkin's "twinning", shares many of its goals with Cake. While their tool supports a strictly smaller set of adaptations than Cake (which does not include the breadth of context-sensitive, stateful or many-to-many mappings for which Cake was designed), Cake lacks its "deep adaptation", meaning the ability to factor two variant components into a single component targeting a more abstract API.

Cake is also a component assembly language. It was influenced by Knit [Reid et al. 2000], but radically extends its adaptation capabilities. It shares its deliberate computational constraints with SuperGlue [McDirmid and Hsieh 2006], but applies to a different programming problem. Other work in component orchestration and coordination, such as Reo [Arbab and Mavaddat 2002] or Orc [Misra and Cook 2006] generally does not include adaptation as a primary goal, and consequently lacks a full feature-set, but sometimes nevertheless caters to some adaptation use-cases in a black-box style similar to Cake's.

Flexible Packaging [DeLine 2001] is perhaps the work with most closely aligned long-term goals to Cake, in seeking to separate functionality from integration, but takes a clean-slate approach. It also concerns only matters of "style" (Cake's analogy with *packaging*) rather than detailed interface mismatch (since it is envisaged that the packaging author would also write code to adapt the details).

9. Conclusions

We have presented the design and implementation of Cake, a language designed to abstract the adaptation, composition and integration of mismatched components by describing abstract *relations* between component interfaces. Our implementation for native binaries finds novel use for debugging information and applies novel techniques to enable dynamic behaviour and selective sharing of objects exchanged by such code. We have demonstrated how Cake's features apply to real coding tasks, and our application of Cake to three real case studies demonstrates its ability to yield simpler, better modularised code.

Acknowledgments

The author is grateful for feedback and encouragement from David Greaves. Amitabha Roy suggested using the on-stack return address to interpose on stack frame cleanup, and provided code. Jamey Sharp provided valuable support for the XCL case study. This version has benefited from helpful suggestions from Michael Hicks, Jon Crowcroft, Tim Deegan, Tim Harris, Orion Hodson, David Greaves, Alan Mycroft, Dominic Orchard, Robin Message, Jukka Lehtosalo, Derek Murray, Steven Hand, Chris Smowton, David Evans, Atanu Ghosh and Alan Lawrence. This work was supported by an EPSRC Doctoral Training grant.

References

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

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.

D. Beazley. Swig: An easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the 4th USENIX Tcl/Tk Workshop*, pages 129–139, 1996.

A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt. Boomerang: resourceful lenses for string data. In *Proc. POPL '08*, pages 407–419. ACM, 2008.

A. Bracciali, A. Brogi, and C. Canal. A formal approach to component adaptation. *J. Syst. Softw.*, 74:45–54, 2005.

G. Bracha, C. Clark, G. Lindstrom, and D. Orr. Module management as a system service. In *OOPSLA Workshop on Object-oriented Reflection and Metalevel Architectures*, 1993.

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

D. Dig, S. Negara, V. Mohindra, and R. Johnson. ReBA: a tool for generating binary adapters for evolving java libraries. In *Proc. 30th Intl. Conf. Softw. Eng.*, pages 963–964. ACM, 2008.

DWARF Debugging Information Format version 3. Free Standards Group, December 2005.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.

W. Harrison and H. Ossher. Subject-oriented programming: a critique of pure objects. *ACM SIGPLAN Not.*, 28:411–428, 1993.

J. Järvi, M. Marcus, and J. Smith. Library composition and adaptation using C++ concepts. In *Proc. 6th Intl. Conf. on Generative Programming and Component Engineering*, pages 73–82, 2007.

A. Kantee. Rump file systems: Kernel code reborn. In *Proceedings of the 2009 USENIX Annual Technical Conference*, Berkeley, CA, USA, 2009. USENIX Association.

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*. ACM, 2005.

S. McDirmid and W. Hsieh. SuperGlue: Component programming with object-oriented signals. In *ECOOP 2006*. Springer, 2006.

J. Misra and W. Cook. Computation orchestration: A basis for wide-area computing. *J. Softw. & Syst. Modeling*, 6:83–110, 2006.

I. Neamtiu and M. Hicks. Safe and timely dynamic updates for multi-threaded programs. In *Proc. PLDI '09*, pages 13–24, 2009.

I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for C. In *Proc. PLDI '06*. ACM, 2006.

M. Nita and D. Notkin. Using twinning to adapt programs to alternative APIs. In *Proc. 32nd Intl. Conf. Softw. Eng.* IEEE, 2010.

D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, 1972.

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

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

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

J. Sharp and B. Massey. XCL: An Xlib compatibility layer for XCB. In *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*. USENIX Association, 2002.

A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *Proc. ESEC-FSE '07*, pages 35–44. ACM, 2007.

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

A. Complete example

```
exists elf_reloc("libmpeg2play.o") mpeg2play;
exists elf_external_sharedlib("avcodec") avcodec;
exists elf_external_sharedlib("avformat") avformat;
exists elf_external_sharedlib("avutil") avutil;
alias any[avcodec, avformat, avutil] ffmpeg;

derive elf_reloc("mpeg2play2ffmpeg.o") program = link[
  mpeg2play, ffmpeg
] {
 mpeg2play ⟷ ffmpeg
 {
  fopen(fname, "rb")[0] ⟶ av_open_input_file(out _, fname);
  values FILE ⟷ AVFormatContext /*{}*/;
  mpeg2_init() ⟶ { avcodec_init();
                   av_register_all() }
                 ⟵
                 {new mpeg2_dec_s};

   let f = fopen(fname, "rb"), ...,
   let dec = mpeg2_init(), ...,
   mpeg2_get_info(dec) ⟶{
          av_find_stream_info(f) // in-place update to f
       ;& let dec...vid_idx = find( // standard algorithm
              f↪streams,
              fn x ⇒ x↪codec↪codec_type // lambda!
                      == CODEC_TYPE_VIDEO )
       ;& let codec_ctxt = f↪streams[dec...vid_idx]
       ;& let codec = avcodec_find_decoder(
                          codec_ctxt↪codec_id)
       ;& avcodec_open(codec_ctxt, codec)
       ;& codec_ctxt };

  values (dec: mpeg2_dec_s, info: mpeg2_info_s,
  sequence: mpeg2_sequence_s, fbuf: mpeg2_fbuf_s)
   ⟷ (ctxt: AVCodecContext, vid_idx: int,
      p: AVPacket, s: AVStream, codec: AVCodec)
  {
    // ensure an AVPacket exists, on any flow L-to-R
    void ⟶?(new AVPacket tie ctxt) p;

    // picture dimensions are in sequence and ctxt
    sequence ⟷ ctxt {
     // width and height done automatically
     display_width ⟵ width;
     display_height ⟵ height; // here we assume a
     chroma_width ⟵width / 2; // 4:2:2 pixel format
     chroma_height ⟵ height / 2; };

    // info.sequence always points to sequence object
    info.sequence (&sequence)⟵? void;

    // special conversion required for buffers
    fbuf ⟷ frame {
     buf[0] as packed_luma_line[height] ptr
       ⟷ data[0] as padded_line[ctxt.height] ptr;
     buf[1] as packed_chroma_line[chroma_height] ptr
       ⟷ data[1] as padded_line[ctxt.height / 2] ptr;
     buf[2] as packed_chroma_line[chroma_height] ptr
       ⟷ data[2] as padded_line[ctxt.height / 2] ptr;
    };
};
values packed_luma_line ⟵padded_line {
 void (memcpy(this, that, display_width))⟵ void; };
values packed_chroma_line ⟵padded_line {
 void (memcpy(this, that, chroma_width))⟵ void; };

/* The loop in ffmpeg proceeds frame-by-frame, whereas in libmpeg2
 * each iteration might yield zero frames (in the STATE_BUFFER
 * case) *or* one or more frames (in the STATE_SLICE case). Solve
 * this by ensuring that each iteration yields exactly one frame---
 * a case supported by both library and client. */
mpeg2_parse(dec)[0] ⟶ { void }
                    ⟵
            STATE_BUFFER;
/* Notice use of [0]: "the first call to mpeg2_parse()
 * *on a given dec, for all dec* returns STATE_BUFFER */

/* Reading from the input file handle must also be mapped to an
 * ffmpeg library call. Since success of fread() entails a return
 * value of nmemb, we must return this, irrespective of the size
 * of the frame actually read. This is a rare example where error-
 * -reporting conventions must be explicitly satisfied in stubs. */

let f = fopen(fname, "rb")[0], ...,
let dec = mpeg2_init(), ...,
fread(_, _, nmemb, f) ⟶
  { { av_read_frame(dec...packet, f) ;& nmemb } ;| 0; };

/* Since ffmpeg handles input buffering for us, no
 * action is required on a call to mpeg2_buffer(). */
mpeg2_buffer(_, /*buf*/_, /*buf + siz*/_) ⟶ { void };

/* The client calls mpeg2_parse() to request decoded frames. This
 * translates to a call to avcodec_decode_video(). Since our earlier
 * call to av_read_frame() may have returned a frame from a
 * different stream (e.g. an audio stream in the same file), we have
 * two cases to consider. These map directly to the libmpeg2 constants
 * STATE_BUFFER ("must read more data") and STATE_SLICE ("one or more
 * decoded frames available"), distinguished by an if--then--else. */
f ⇐ fopen(fname, "rb")[0], ...,
dec ⇐ mpeg2_init(), ...,
size ⇐ fread(_, _, nmemb, f),
mpeg2_parse(dec) ⟶{ let frame_avail = (
  if dec...packet.stream_index == dec...vid_idx
  then { av_free(dec...frame); // this is null-safe
         let dec...frame = avcodec_alloc_frame();
         avcodec_decode_video2(dec...codec_ctxt,
           frame, out got_picture, dec...packet);
         true }
  else false )
  }--
  ⟵
  --{ if frame_avail then STATE_SLICE
                    else STATE_BUFFER };
/* Notice the special reverse-arrow syntax for returning. Moreover,
 * the special "--{" ("continuing") syntax retains all name bindings
 * from the preceding stub. */

/* Finally, we relate the state tear-down calls of the two interfaces. */
mpeg2_close(dec) ⟶{ av_free(dec...picture);
                    avcodec_close(dec...codec);
                    av_close_input_file(dec...ic); }
                 ⟵
  { delete dec };
 } // end mpeg2play ⟷ ffmpeg
}; // end derive
```