The Mythical Matched Modules

Overcoming the tyranny of inflexible software construction

Stephen Kell

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

Abstract

Conventional tools yield expensive and inflexible software. By requiring that software be structured as plug-compatible modules, tools preclude out-of-order development; by treating interoperation of languages as rare, adoption of innovations is inhibited. I propose that a solution must radically separate the concern of *integration* in software: firstly by using novel tools specialised towards integration (the "integration domain"), and secondly by prohibiting use of preexisting interfaces ("interface hiding") outside that domain.

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

General Terms Languages

1. Introduction

Software is expensive: expensive to develop, and expensive to modify or change because of its inherent *inflexibility*. By the latter I refer both to the difficulty of *maintaining* software and also to software's tendency to *grow in silos*. Software grows as islands of functionality, founded on infrastructure including programming languages, UI toolkits, development "frameworks", extensible applications (browsers like Firefox, editors like Emacs) and so on. The same functionality is frequently found replicated across many silos of a given class, at considerable expense—this is strong evidence for the underlying inflexibility of software. To fix these problems requires changes to both *tools* and *practices*—where these two are highly interdependent.

OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA.
Copyright © 2009 ACM 978-1-60558-768-4/09/10. . . $10.00

Toolchains—encompassing compilers, linkers, stub generators and more—have grown organically into a design which informs much of common practice. However, this design originates from an idealised conception of development as an activity with central planning (enabling coordination of interfaces among cooperating modules), linear progress (enabling *depended-upon* interfaces to be finalised before their dependents are coded) and perfect anticipation (ensuring that earlier decisions need never be reversed). Unfortunately, the resulting tools embody several false assumptions, giving them designs *optimised* towards building inflexible, siloed software, and which *penalise* the development of flexible and easily-interoperable software. However, when considered from afar, these assumptions are absurd. Specifically, I identify and refute the following latent assumptions:

- that software is structured as *plug-compatible* modules, grown "in order" from low-level dependencies upwards;
- that language interoperation is an exceptional case;
- that information hiding is a sufficient strategy against *coupling*.

I will argue that simpler and more flexible software is possible with tools designed in conscious avoidance of these assumptions. Specifically, I propose a radical two-step change in tools and practices of software construction:

- an *integration domain*—there is a fundamental need for languages and tools specialised towards composition of software, and moreover, these should *not* resemble conventional languages;
- the practice of *interface hiding*—information hiding should be extended to a new level in which components are constructed while *purposely and completely ignoring* the interfaces of all foreign components, and where this is enforced by tools.

I begin by reviewing the concept of the *integration concern*, which will underpin most of the subsequent discussion.

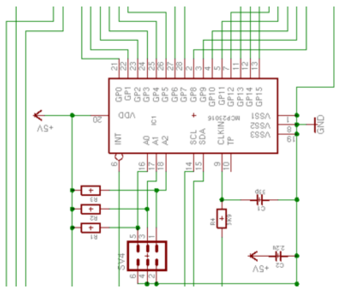
2. The integration concern

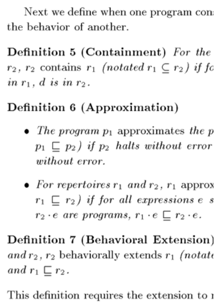
All useful software has some interaction with its environment. Even if simply printing out a response, returning an exit status, or adjusting some output signal, software is never an island. As such, software incorporates knowledge, or assumptions, about the nature of its environment—what interaction mechanisms are available, and what conventions are to be adopted in their use. Mechanisms might include procedure calls or byte-stream communication, and conventions include the signatures of those procedures or the syntax of the valid byte-stream exchanges (together with semantics in both cases). It is not possible to interact without some such assumptions; these are a fundamental and recurring hindrance to software, in that they are precisely the source of *coupling*.

The same functionality can be useful in many different environments. For example, a spell checker might be somehow useful in a word processor, an e-mail application, a batch document processor or a voice recognition engine. Of course, the developer *must* inescapably assume *some* environment in order to write code. The common result is software that is strongly coupled to one specific environment—consisting of whatever pre-existing components the developer preferred or knew about. Consequently, reimplementation abounds: similar functionality is replicated for each operating system, each programming language, each desktop environment, each text editor and so on.[1]

I refer to code's embodied knowledge or assumptions about the environment as *integration details*, and to their collected intent within a piece of software as the *integration concern*. To avoid the expense of reimplementation, we would clearly like to *separate* the concern of integration, by modularising integration details separately from functionality as far as possible. This will lessen the problem of coupling and improve the flexibility of our software. However, in common practice this has only been pursued to a limited extent (albeit an important one) in the practice of *information hiding* [Parnas 1972]. This is a *coupling minimisation* technique: it limits *how many* decisions concerning the environment are visible to a module. However, we have established that *some* coupling is unavoidable. There has been relatively far less progress on how to *mitigate*, through tools and languages, whatever coupling cannot be eliminated. (As I will argue, what work does exist either provides only a *clean-slate* solution or presents inadequate abstractions.)

In domains other than software, separation of the integration concern is already established practice. Figure 1 illustrates two of these pictorially. In circuit design, engineers do not expect that their integrated circuits to be wired together directly with other ICs. Instead there is a whole vocabulary of glue components, including resistors and capacitors and small logic arrays, constituting a separate integration domain. Separating the integration concern simplifies the IC's design, reduces cost and improves flexibility with respect to other use contexts.[2] As a second example, consider that in technical writing authors invariably define their own terminology up-front without any obligation for consistency with other authors.[3] The primary concern is to choose a set of definitions *convenient for the author's work*, meaning one which make the work precise and comprehensible. The concern of integration is addressed separately—the reader, when reviewing and comparing different pieces, is well accustomed to the need to relate differing definitions.





Next we define when one program con

the behavior of another.

Definition 5 (Containment) *For the*
r_2, r_2 contains r_1 *(notated $r_1 \subseteq r_2$) if fo*
in r_1, d is in r_2.

Definition 6 (Approximation)

- *The program p_1* approximates *the p*
$p_1 \sqsubseteq p_2$) if p_2 halts without error
without error.
- *For repertoires r_1 and r_2, r_1* approx
$r_1 \sqsubseteq r_2$) if for all expressions e s
$r_2 \cdot e$ are programs, $r_1 \cdot e \sqsubseteq r_2 \cdot e$.

Definition 7 (Behavioral Extension)
and r_2, r_2 behaviorally extends *r_1 (notate*
and $r_1 \sqsubseteq r_2$.

This definition requires the extension to

Figure 1. Integration in circuit design and technical writing

Given these precedents, the ways of software seem bizarre. As with separation of concerns generally, practice is highly dependent on tool support, which conventionally is extremely limited [Tarr et al. 1999]. In the following section I will examine the assumptions in existing toolchains and resulting practices. (Note that I am describing assumptions *latent in the design of tools*—not necessarily in the mind of any particular developer.)

3. Myths and realities

3.1 Plug-compatibility and in-order development

Toolchains implicitly assume that software consists of modules whose interfaces match directly and precisely. This depends on the lemma that code *providing* some functionality *exists before* the code *requiring* that functionality—at least to the extent that its interface is known. For example, when using libraries, clients of a library are written after the interface of that library is determined, so that they can target that interface in a *plug-compatible* way.[4] The in-order myth is subject to the classic caveat that it works only if we make correct judgements *ab initio* which continue to hold

[1] Meanwhile, it also seems likely that a significant proportion of the software industry is occupied by development of web interfaces, billing systems, business process models and various other common projects, each instance overlapping substantially in functionality.

[2] It is no coincidence that with ICs there is a very tangible cost to on-chip complexity, whereas in software the cost of added complexity from mixing integration concerns appears more abstractly (in payroll, downtime, etc.).

[3] Although agreement is of course helpful, writers would find it far too restrictive to be limited to preexisting definitions.

[4] Usually interfaces change often enough through early evolution of a component's implementation that in practice, much of the target component must exist before a stable implementation of any client can be produced.

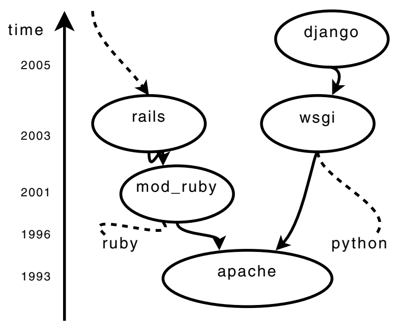


Figure 2. In-order growth of software stacks.

over time. It is clearly falsified by *evolution* of interfaces, by *decentralised development* (essentially evolution in parallel) and by *porting* or retrofitting, which inherently concern *incompatible* interfaces.

Cost Retrofitting tasks may seem rare, but they do happen. To pick one visible example, the entire KDE desktop suite recently migrated from the DCOP IPC system to D-BUS[5], at huge effort. DCOP details pervaded the source code of KDE applications, entailing huge changes. (It is also interesting that the standardisation effort was motivated by the desire to interoperate between previously siloed applications, such as the GNOME desktop suite.) Tellingly, the less desirable alternative to porting, namely reimplementation, is anecdotally more common—despite obvious disadvantages.

Research perspectives Previous authors have noted the in-order assumption, sometimes called *provider–consumer asymmetry* [Reid et al. 2000, Arbab and Mavaddat 2002]. In-order development is common because targetting existing concrete interfaces is the simplest and most expedient structuring technique—but there are others. Designs featuring intermediate abstraction layers are often devised, for example in libraries where multiple back-ends are anticipated.[6] These designs are inherently more resilient to changes in depended-upon code. It may therefore seem unreasonable to cite in-orderedness as a weakness of toolchains, when with more foresight a better design can be produced using current tools. However, foresight is rare. Better tool support can enable better-abstracted designs *naturally*, by default, rather than as a special case requiring up-front effort which is rarely made. The key strategy, already proposed in prior research work [DeLine 2001] is a division of responsibility in toolchain design between functionality and integration.

Vision Current practice is not our only concern. Not only might better tool support simplify existing development scenarios, or enable more successful instances of decentralised development or porting. More boldly, I claim it can also

[5] http://dbus.freedesktop.org/

[6] Textbook examples include cross-platform windowing toolkits such as wxWidgets, http://www.wxwidgets.org/ or Java's SWT, http://eclipse.org/swt.

```
C         extern int Foo_foo(char *);
Java      foo.Foo f = new foo.Foo(); // implicit
Haskell   import Foo(foo)
Python    from foo import Foo
```

Figure 3. Module imports implying homogeneity.

cause paradigm shifts. Placing high-level tool support for integration and adaptation *close to the user*, for example within web application mashup platforms and browser extensions[7] has already led to added-value innovations[8]) which could not have been anticipated by the creators of the underlying software. In my ultimate vision, the techniques I propose here could enable the same for potentially *all* software—experienced users could straightforwardly mash together entire applications or pieces thereof, to meet their desired functionality, in a fuller realisation of the elusive Unix vision of user-defined compositions of functionality.

3.2 Homogeneity

Figure 3 shows declarations of external modules in several different languages. In each case the foreign module exposes equivalent functionality. Implicitly, that module must be written in the same language. Toolchain support when this is *not* the case comes as an ad-hoc selection of intricate and unwieldy "language interoperability" features. As I will argue, their unwieldiness persists because of an unstated assumption that such requirements are a rare *exceptional case*.

Cost Are such requirements really rare? New programming languages continue to emerge, and are seen as a promising longer-term solution to the expense of software. However, adoption by practitioners lags far behind the state of the art. Other authors have already speculated on the reasons for this [Wadler 1998], but one key factor is the difficulty of *incremental* adoption. This is essential because it is rare to write an entire application from scratch. Unfortunately, conventional treatments of inter-language linking have several weaknesses, rooted in the "exceptional case" assumption, which discourage multi-language development.

Firstly, there is *no encapsulation of language decisions*. If a foreign module is *not* defined in the same language, any *import* statements must betray this—for example, in Java the native keyword is required, in C an extern declaration, and in Haskell a foreign import declaration. This clearly a failure of information hiding, yet is questioned little because of the *exceptional case* myth.

Secondly, language interoperability schemes such as Java's JNI [Liang 1999] or Haskell's FFI [Chakravarty et al. 2002] are convoluted from the desire for a *universal map-*

[7] e.g. Mozilla Ubiquity, http://ubiquity.mozilla.com/, Greasemonkey, http://www.greasespot.net/

[8] e.g. http://www.shiftspace.org/, http://www.housingmaps.com/

ping between interface definitions in two languages—that is, one defined for *all interfaces* and implementable for *all possible implementations* allowed by those languages. This has little benefit for the programmer, who is working with specific interfaces and, in practice, probably interested in only a subset of conceivable language implementations. (An interesting departure from universality is in the GNU implementation of Java [Bothner 2003]. By foregoing universality, GCC provides CNI, a much more natural alternative to JNI.)

Thirdly, C ia nearly always chosen as the unifying medium for expressing glue logic. A unifying medium is clearly useful because it converts problem of size n^2—mapping all languages to all other languages—into one apparently of size $2n$. However, C is a poor unifier: manual resource management, reliance on mutable storage and machine- and compiler-dictated object layout present a low-level medium for glue code, while the lack of run-time checks greatly complicates debugging. Joining two higher-level languages together is accordingly often special-cased (e.g. Jython[9]).

Finally, it is telling that if some new language X *is* adopted, it is always accompanied by considerable reimplementation of tool functionality "for X". Consider that practically every programming language has a lex-like tool. Why should this be? A lex-generated lexer, being a separate module from its client, should be invokable from any client language, with the language of the generated code an encapsulated concern.[10]

Research perspectives Much research work concentrates on *theoretical* aspects of interoperability, by asking how *reasoning mechanisms*, such as type systems and run-time checks, can be extended to preserve their guarantees in the presence of foreign-language modules [Wadler and Findler 2009]. This work is valuable but is not our concern here.

Existing *practical* approaches are largely found in implementations. Microsoft's CLR [Meijer 2002] defines an intermediate abstraction, roughly par with an object-oriented garbage-collected language, and having standardised data representations. This is a reasonable but *clean-slate* approach: it excludes all code lying outside that standard.

Generative tools are available to ease the task of creating JNI and similar glue code. Wrapper generators such as JunC++tion [11] generate proxies presenting a more natural interface than that of the underlying interoperability interface (in this case JNI). Swig [Beazley 1996] provides an annotation language for customising the generated proxy's interface, where annotations correspond to macros expanded within the C or C++ glue. These tools demonstrate the feasibility of taking code in some fairly relaxed *natural style* and then, in a separate step, adapting that code to fit a different interface. However, both approaches are limited: to specific languages (one side restricted C or C++) and in their flexibility (either none, or in the case of Swig, to definition of new "typemap" macros—a highly involved task, owing to the brittleness of macro expansions, the potential for feature interactions, and so on).

Vision The homogeneity assumption may appear unavoidable. One property of a language is that it defines a model of some universe. Therefore, perhaps by definition it cannot express references to artifacts that lie outside that universe, in some other language. I call this the "model problem", but it is not actually a problem. It is overcome by the ability to *interpret* one piece of code *as if it were something else*—where defining interpretations of foreign code is a fundamental requirement of integration. The "universal mappings" of interoperability schemes are simply one interpretation from a wide space—a space we would like to open up fully to the programmer. This concept of *subjectivity* [Harrison and Ossher 1993, Batory] appears throughout software.[12]

Referencing some foreign entity within code does not determine *what that entity should be* but only *how it must appear*. At present, toolchains[13] do not support the interposition necessary to effect this transformation from *what* to *how*—compilers and linkers understand only one view of any given code, and tool support for *defining* and *transforming* views is lacking. The scope for such support is huge. Consider a command-line tool with its execve() interface. One interpretation of this interface shows execve()'s arguments as arrays of strings. Another shows them as a disjoint union of options and typed arguments, reflecting the command's syntax. Tool support must permit stackable descriptions of these interpretations, each described in terms of lower-level conversions and rooted in primitives akin to atoi(). (As I will describe in Section 4, *relations* are one convenient abstraction for describing these and other instances of subjectivity.)

3.3 Doing better than information hiding

Information hiding [Parnas 1972] is rightly considered a fundamental strategy for reducing coupling between modules. It is supported by all contemporary programming languages. As described in Section 4, it *minimises* coupling by limiting the visibility of implementation decisions between modules. However, I have described how *some coupling is inevitable*. Conventional languages do nothing to mitigate that coupling: on whatever implementation decisions *are* exposed, modules are expected to agree. Where incompatibili-

[9] http://www.jython.org/

[10] I chose lex not yacc because lex has less need for semantic actions—but the continued popularity of these tools' designs is a mystery, given how they advocate such a thorough mixing of application logic with the concern of language recognition. This contrasts with more modern approaches like Antlr [Parr and Quong 1995]).

[11] http://codemesh.com/products/junction/

[12] Network filesystems are a classic example in operating systems, and the adaptor pattern [Gamma et al. 1995] is well known in object-oriented programming.

[13] . . . and also runtime systems, including operating systems.

ties arise, the only recourse is to edit the code to regain compatibility, or to code an adaptor.[14]

Cost When the plug-compatibility myth fails—for example in porting tasks—any coupling to an incorrect environment must somehow be worked around. Invasive editing of code is common, but yields a fork or patchset which is highly syntactic and inherently fragile. Black-box adaptors are more modular, but are labour-intensive to create and, while less fragile, still nontrivial to maintain. Sometimes a mixture of both techniques is used—for example in the KDE migration mentioned in Section 3.1, a black-box abstraction layer (QtDBus) was designed to resemble the earlier DCOP API, while still differing both abstractly in a few ways and concretely in many (including most function names).

Vision To continue the example, if KDE source code had been kept abstract and avoided embedding DCOP-specific details, invasive porting would have proved unnecessary. Perhaps the developers proceeded by *imagining* that some minimally sufficient IPC interface was available, ignoring the issue of whether any implementation was available until later. Subsequently, a separate layer of software, mapping abstract KDE references to concrete DCOP or D-BUS references, would be required. As with information hiding, such a design requires *discipline* to keep the original code abstract. Just as tools *enforce* information hiding, so can tools enforce this stronger sense of modularity—by *preventing the import of any concrete interface*, and using integration tools to bridge the resulting gap between the abstract and the concrete. The remainder of this paper discusses techniques for realising this two-stage approach.

4. The integration domain—what and why

An integration domain is simply a set of languages or tools for performing integration of software. Informally, many ad-hoc special-purpose integration domains have emerged in conventional practice—for example scripting languages (like the Unix shell), patching tools (like Unix's diff) and stub generators (as in various RPC implementations [Birrell and Nelson 1984]). Each of these is highly specialised and highly constrained, but hint at the more general and powerful tools which my arguments so far have motivated. Two questions remain: why are conventional programming languages not sufficient for this domain, and what form should the alternative take?

4.1 An example: the Cake linking language

I begin by giving an example from my ongoing work of one tool, the Cake linking language compiler, which fulfils a small part of the vision I have outlined. It is described more fully in an earlier short paper [Kell 2009]. Figure 4 shows a fragment of Cake code relating function calls and the values they exchange across a mismatched interface.

[14] These are *white-box* and *black-box* approaches, respectively.

```
switch ↔ libgtk20 { /* old interface ↔ new interface */
 gtk_window_set_policy (win, shrink, grow, _) →
  ( if shrink then gtk_window_set_size_request (win, 0, 0) else void;
    if grow   then gtk_window_set_resizable (win, TRUE) else void );

 (preview_window ::) gtk_signal_connect_object (i, d, c_h, data) →
   g_signal_connect_data (i, d, c_h, data, null, {});

 values {
  GtkWindow ↔ GtkWindow {
   type as GtkWindowType ↔ type as GtkWindowType;

   window_has_focus ↔ has_focus;
    auto_shrink ← const 0;

    use_uposition → need_default_position ;
    use_uposition → need_default_size ;
    use_uposition ← ( need_default_position || need_default_size );
}}}
```

Figure 4. Relating function calls and data structures in Cake

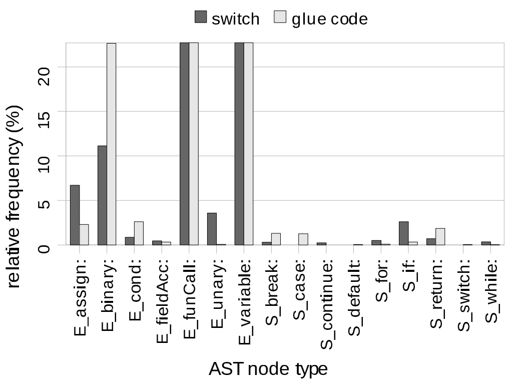
The details of the code are not important here, but I draw attention to several properties of the language. It is adoptable, largely because it chooses a unifying abstraction already satisfied by substantial existing codebases (namely relocatable object code). It complements component programming languages: plug-compatible compositions can trivially be expressed as "no-op" Cake descriptions similar to linker invocations in a makefile. Once plug-compatiblity is violated or retrofitting is desired, Cake's adaptation features can be invoked. It is high-level, declarative and minimal: it simply expresses relations between runtime values, optionally predicated on the context in which they occur (including function calls, but extending to call contexts, call sequences, surrounding data structures and beyond).

Why are these good properties? What makes Cake better than a conventional programming language for integration tasks? I now address these questions.

4.2 Why an integration domain?

Glue code is different from other code. The split has been characterised in many ways: unstable versus stable [Nierstrasz and Achermann 2000], coordination versus computation [Arbab and Mavaddat 2002], or functionality versus packaging [DeLine 2001]. Here I argue specifically that *alternative languages* are not only useful for notationally *separating* the integration concern, but are *essential* for maximally abstracting it.

Expressivity and abstraction Component programming languages are a poor notation for adaptation tasks; they invariably offer the wrong abstractions or the wrong level of abstraction. Typically, adaptation logic is algorithmically simple. It defines few or no new data types, few new functions except for wrappers around existing ones, and likely makes relatively little use of looping or recursion. Rather, it is concerned with *recognising* and *relating* the interac-



(Note: E_variable and E_funCall extend off the chart.)

Figure 5. AST node frequencies in glue and non-glue code

tions defined by two *existing* mismatched interfaces. When written in conventional languages, it most often contains the following kinds of code: case analysis (pattern-matching, switch statements or if–then–else), data movement (by assignment or parameter passing), multiplexing and demultiplexing, look-up tables, dynamic mappings (like dictionaries or other key-value stores), simple computations (for re-encoding of messages; usually already found in libraries), simple buffering, state machines, and (sometimes) concurrency control operations (like fork–join patterns, thread wait conditions, and so on).

A simple experiment lends some empirical credence to this assertion. Figure 5 shows relative frequencies of various syntactic features in C and C++ code from the gtk-theme-switch Cake case study [Kell 2009]. One series measures the glue code (∼8000 lines, partially autogenerated), the other the main source code of the application (∼1000 lines). Although the data is far from conclusive, notice how loop constructs are relatively far less common in glue code, while switch and case statements appear more often. (The lower glue frequency of if and assignments is perhaps explained by the skewed nature of the adaptations required by this case study; clearly further experiments would be helpful.)

Unfortunately, these are mostly features for which conventional languages are not optimised towards abstracting.[15] Consider a language such as Java—possibly a good choice for coding some set of components, but it may be poor for adapting those same components, perhaps because it lacks pattern-matching or curried functions (for demultiplexing function calls). Meanwhile, a language such as Haskell, while having these features, could be awkward if, say, some previously stateless adaptation logic was to be made stateful by addition of a state machine or dynamic map. (This is a complex change to realise in Haskell because adding state requires profound changes to type annotations.) The argument here is not that these are necessarily weaknesses in the languages, but merely that the languages were designed for abstraction of algorithms and data structures, and not abstractions of *relations* among messages or interactions.

Automation and reasoning Being computationally simpler than other code, glue code is potentially more tractable for purposes of automatic reasoning and even automatic synthesis. Protocol adaptation—one of the most well-explored kinds of adaptation [Yellin and Strom 1997, Passerone et al. 2002, Reussner 2003, Bracciali et al. 2005]—is invariably implemented using only finite-state abstractions (either interface automata or replication-free pi calculus). This is strictly less powerful than a Turing-complete abstraction, but appears to be sufficiently useful for a very wide range of tasks. Crucially, its lesser computational complexity makes it more amenable to automatic reasoning—hence the automatic or semi-automatic nature of the cited tools. In general this suggests that the computational power required in glue logic is, in a reasonable proportion of cases, likely to be less than Turing-powerful.[16] Future work will no doubt increase the space of synthesisable adaptation logic. Although it is likely that much of the overall space will remain essentially a manual task (but still with the potential for better abstraction through improved languages), keeping integration details separately modularised, and expressed in a purpose-built notation, allows for constraining the computational power of the notation and increases the potential for identifying and implementing synthesis and/or verification techniques.

4.3 What: relations, not scripts or circuits

Research work has proposed several new integration domains: coordination languages [Arbab and Mavaddat 2002], "delta" languages [Keller and Holzle 1998] and semantic patching tools [Padioleau et al. 2008], linking languages [Reid et al. 2000], orchestration languages [Misra and Cook 2006] and formally grounded scripting languages [Achermann and Nierstrasz 2001]. Only time and experience can truly prove the worth of these. However, scripting-style notations alone are clearly limited, because they are so similar to conventional languages. Dynamism alone cannot abstract the integration domain; meanwhile, the similarity will likely lead to *leakage* of application concerns into integration logic. Deltas and patches suffer from a restrictive "second class" conceptual asymmetry—one cannot easily apply a delta to a delta, say. Data-flow networks as in Reo, while useful for scenarios where concurrency is paramount, are surprisingly difficult to design or explain[17] and currently of-

[15] There is similarity between this set of programmatic features and the logic contained within a table-driven parser. Both are also examples of code which are far more suited to being generated from a higher-level representation than being coded in a conventional language. Although recursive descent, meanwhile, is a reasonable fit for conventional languages, choosing recursion as the means of expressing the grammar is far more constraining.

[16] The kinds of additional specifications required by these techniques are already being incorporated into practical tools, and their benefit is likely to outweigh any burden.[Barnett et al. 2005, Flanagan et al.]

[17] Testament to this is the difficulty faced by Clarke et al [Clarke et al. 2007] in concisely explaining the operation of a very simple exclusive

fer no means of expressing simple data-dependent behaviour (for example, rearranging the fields in a structured message).

The choice of relations, as exemplified by Cake, has several benefits. Simple relations on values are expressed intuitively as pattern-matching. More complex relations can be built up by incorporating contextual guards in patterns: contexts may be either spatial (e.g. relations applying only to values within certain function calls, or at certain points within a larger data structure) and temporal (relations specialised towards values as they occur within particular sequences of function call exchanges—this is protocol adaptation). Analogously with grammars, greater context dependency can bring arbitrarily higher expressivity, but one hopes that a practically useful level of expressivity can be reached using only a very restricted degree of context dependency.

5. Interface hiding

Suppose an expressive integration domain is available. What strategy might best avoid coupling? My suggestion, strange as it may seem, is to avoid directly importing any foreign interface whatsoever. Similar to the integrated circuits example in Section 2, each component may define its own interface to the outside, designed to keep the component simple and comprehensible. It is a separate task, supported by integration tools, to glue components into a functional ensemble.

5.1 Motivational experiment—complexity inheritance

Latent in this argument is the assumption that targetting concrete interfaces increases the complexity of code. Software depending on some concrete interface is clearly moulded by the details of that interface; a more general or complex interface may force the client to incorporate unwanted complexity, even if that client exercises only a small subset of the library's functionality. If so, the client *inherits* the complexity of the interface it targets. I conducted a small experiment to demonstrate this phenomenon.

I used open source libraries to perform two different tasks—one storage-oriented (writing a persistent key-value set describing the program memory map) and one computational (video decoding). In each task the same functionality was implemented twice, using different C libraries—the second chosen to be substantially more complex (and more general) than the first, yet offering a similar level of abstraction. The library pairs were firstly libmpeg2 and ffmpeg (comprising libavcodec and libavformat) and secondly Berkeley DB and sqlite's BTree interface. Table 1 presents simple summary measurements of the libraries' public interfaces, the API subset (or "slice") exercised by the client, and the clients' source code. The key observation is that the client of the more complex library is itself more complex.

router circuit (page 4). The unintuitive arrangement of channels is said to "conspire to" ensure exclusivity.

	mpeg2	ffmpeg	db	sqlite
API function count	24	151	208	214
mean signature size[3]	3.3	4.3	2.8	4.3
API structures count	10	26	25[2]	12
mean structure size[4]	5.6	19	17[2]	8.0
slice function count	5	14	5	12
mean signature size	2.8	3.1	6	4
slice structures count	3	6	2	4
client size (LoC)	71	93	51	75
client API calls	5	14	5	13
helper calls[5]	2	1	6	3
mean call size	3.4	3.2	4.9	3.9

[2] imprecise owing to undocumented public–private division
[3] signature size $= 1+$ number of in or out parameters
[4] simple total of all public fields
[5] C library calls necessitated indirectly by API usage

Table 1. Measurement of complexity inheritance

5.2 Mitigating complexity

Using interface hiding, complexity inheritance clearly would not occur. Instead, the additional complexity would appear in the integration domain. As I described in Section 4.3, there are two major reasons why this complexity can be better handled from such a domain: firstly, the availability of higher-level notations tailored to the task of *relating* corresponding interface elements, more suitable than glue code; and secondly, the potential for automatic or semi-automatic synthesis of these descriptions.

5.3 Additional benefits of interface hiding

A second payoff of interface hiding is in comprehensibility of client code. Since complexity is not inherited, substantially more readable code may result. Separately, interface hiding forces an explicit statement of *requirements*. Symmetric with explicit *provides* interfaces, *requires* interfaces have been advocated by several component-oriented programming practices [Councill and Heineman 2001] but have yet to appear in conventional practice.

Interface hiding does not require extensive changes to any programming languages. Certain languages—including C and C++—already allow foreign interfaces to be explicitly defined (e.g. by defining new prototypes rather than doing an #include of existing ones). In other languages, some small changes will be necessary to separate the concepts of *imports* (implying a foreign component already exists) from a *declaration* of a view onto the outside world.

Tools also require the *teeth* to *enforce* interface hiding. Analogously to the enforcement of private and protected modifiers, this requires prohibiting imports of concrete external interfaces (except where explicitly overridden).

Clearly, there is a limit to how far interface hiding can be taken. Perhaps truly ubiquitous interfaces, like POSIX, should not be hidden. There is also a risk that the programmer will design an external interface which cannot be satisfied by any foreign components, or which unduly strains the expressivity of the integration domain. In practice, no doubt some iteration will be required to find the optimal modularisation of logic between the integration domain and component internals. These, and other practical questions, must be the subject of future work.

6. Conclusions

I have motivated a two-pronged redesign of software tools and practices with the goal of drastically improving the flexibility of software, and ultimately reducing its cost. Firstly, we require an *integration domain*—tools and languages specially designed for integration of mismatched software, and different from conventional languages. Secondly, we require practices which exploit these tools to maximise separation of the integration concern—*interface hiding* exploits the integration domain to reduce complexity of components and improve their flexibility.

Acknowledgments

I am grateful to D.J. Greaves, Andy Warfield, Yvonne Coady, Derek Murray, Alex Gurney and Alan Mycroft.

References

F Achermann and O Nierstrasz. Applications = components + scripts. In *Software Architectures and Component Technology*, pages 261–292. Kluwer, 2001.

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

M. Barnett, K.R.M. Leino, and W. Schulte. The Spec# programming system: An overview. *Lecture Notes in Computer Science*, 3362:49–69, 2005.

Don Batory. Subjectivity and GenVoca generators. In *ICSR '96*.

DM 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.

AD Birrell and BJ Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2:39–59, 1984.

P Bothner. Compiling Java with GCJ. *Linux Journal*, 2003.

A Bracciali, A Brogi, and C Canal. A formal approach to component adaptation. *The Journal of Systems & Software*, 74:45–54, 2005.

M Chakravarty, S Finne, F Henderson, M Kowalczyk, D Leijen, S Marlow, E Meijer, and S Panne. The Haskell 98 foreign function interface 1.0: an addendum to the Haskell 98 report, 2002. URL `www.cse.unsw.edu.au/%7echak/haskell/ffi/`.

D. Clarke, D. Costa, and F. Arbab. Connector colouring I: Synchronisation and context dependency. *Science of Computer Programming*, 66(3):205–225, 2007.

B Councill and GT Heineman. Definition of a software component and its elements. In *Component-based software engineering: putting the pieces together*, pages 5–19. Addison Wesley, 2001.

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

Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. *SIGPLAN Not.*, 37(5).

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 Notices*, 28:411–428, 1993.

Stephen Kell. Configuration and adaptation of binary software components. In *Proceedings of the 31st International Conference in Software Engineering*, May 2009.

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

S Liang. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley Professional, 1999.

E Meijer. Technical overview of the common language runtime. *language*, 29:7, 2002.

J Misra and WR Cook. Computation orchestration: A basis for wide-area computing. *Journal of Software and Systems Modeling*, 6:83–110, 2006.

O Nierstrasz and F Achermann. Separation of concerns through unification of concepts. In *ECOOP 2000 Workshop on Aspects & Dimensions of Concerns*, 2000.

Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in Linux device drivers. In *Proc. 3rd ACM SIGOPS/EuroSys European Conference*, pages 247–260. ACM, 2008.

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

T.J. Parr and R.W. Quong. ANTLR: A predicated-LL (k) parser generator. *Software-Practice and Experience*, 25(7):789–810, 1995.

R Passerone, L de Alfaro, TA Henzinger, and AL Sangiovanni-Vincentelli. Convertibility verification and converter synthesis: Two faces of the same coin. In *Proceedings of the International Conference on Computer-Aided Design 2002*, 2002.

Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. Knit: Component composition for systems software. In *Proc. of the 4th Operating Systems Design and Implementation (OSDI)*, pages 347–360, 2000.

RH Reussner. Automatic component protocol adaptation with the CoConut/J tool suite. *Future Generation Computer Systems*, 19: 627–639, 2003.

Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, pages 107–119. ACM, 1999.

P Wadler. Why no one uses functional languages. *ACM SIGPLAN Notices*, 33:23–27, 1998.

P. Wadler and R.B. Findler. Well-typed programs can't be blamed. In *ESOP 2009*, page 1. Springer-Verlag New York Inc, 2009.

DM Yellin and RE Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19:292–333, 1997.