Composing heterogeneous software with style

Stephen Kell*
Computer Laboratory, University of Cambridge
Stephen.Kell@cl.cam.ac.uk

ABSTRACT

Tools for composing software impose *homogeneity* requirements on what is composed—that modules must share a language, target the same libraries, or share other conventions. This inhibits cross-language and cross-infrastructure composition. We observe that a *unifying* representation of software turns heterogeneity of components into a matter of *styles*: recurring interface patterns that cross-cut large numbers of codebases. We sketch a rule-based language for capturing styles independently of composition context, and describe how it applies in two example scenarios.

1. INTRODUCTION

Our ability to build software compositionally from unmodified components is limited by two problems. Firstly, tools (such as compilers and linkers) require that composed modules be *plug-compatible*—their interfaces match "in the small". Where this does not hold, compositions are achieved only by laborious glue coding or invasive editing. Secondly, they must be *homogeneous*—functionally compatible modules can not be composed if they are written in different languages, using different interface conventions, different coding styles, or different support libraries. This severely limits the space of possible compositions. Considerable prior work has targeted the first problem [7, 15–17, 21, 23]. However, the second has received only narrow special-case treatments (such as pairwise interoperation between languages [3, 4, 6] or realisations of procedure- or message-based interaction [2, 9]) or clean-slate approaches [10].

This paper outlines ongoing work on an approach to heterogeneous composition, based on *interface styles*. Its key insight is that given an appropriate *unifying medium*—an intermediate representation capturing diverse components—heterogeneity is reduced to differing patterns of usage within that medium, which we call *stylistic variation*. A style is any recurring convention used to realise some programmatic concern; example concerns include error-handling, function dispatch, representation of common data types (lists, sets, strings, etc.), memory management, and so on. Styles are by definition cross-cutting: they recur across large populations of components (i.e. the components that are homogeneous with respect to the style) which may be dissimilar otherwise. By allowing programmers to describe styles abstractly, independent of composition context, we can re-use this descriptive effort to simplify large numbers of composition tasks.

In our approach, styles are described by the programmer using high-level rules. From such descriptions, guided by a set of input components annotated with references to their styles, a glue code generator can compose heterogeneous software essentially by inserting code to "undo" one stylistic concretion and "replay" a different one at the boundary between modules. Our approach is *black-box*, meaning it is sensitive only to an interface abstraction of components. It is also incrementally *adoptable*, in that it applies to a large population of existing components.

Specifically, we present the following contributions:

- we characterise the phenomenon of stylistic variation, by identifying a selection of stylistic concerns and some familiar concretions of each;
- we sketch a notation for describing styles, as an extension to the Cake linking language [16], and present two examples of composition tasks handled using styles;
- we discuss some semantic and practical questions arising, and outline possible future directions.

We begin with a simple example of stylistic variation.

2. CHARACTERISING STYLES

Suppose two programmers independently develop a simple component for counting the lines, words and characters in a file. Fig. 1 shows what they might write. The components are abstractly equivalent, but concretely different. Our goal is to capture these concrete differences programmatically, hence allowing a tool to abstract them away, so that code targetting one of them could instead compose with the other.

We can observe some dimensions of stylistic variation at a glance. Output parameters have been encoded differently, as have character strings. One component provides an explicit resource management API, implicitly also handling initialization and finalization, whereas the other provides only explicit initialization. Naming conventions for multi-word

*The author is now primarily affiliated with the Department of Computer Science, University of Oxford. The details shown remain valid.

FREECO'11, July 26, 2011, Lancaster, UK.

```
struct wc; // implemented in C

// struct is treated opaquely by client

struct wc *word_counter_new(const char *filename);
  // returns NULL and sets errno on error


int word_counter_get_words(struct wc *obj);
int word_counter_get_characters(struct wc *obj);
int word_counter_get_lines(struct wc *obj);
int word_counter_get_all(struct wc *obj,
  int *words_out, int *characters_out, int *lines_out);

void word_counter_free(struct wc* obj);
```

```
class WordCounter // implemented in Java
{
  /* fields not shown... */

  public WordCounter(String filename)
    throws IOException { /* ... */ }


  public int getWords() { /* ... */ }
  public int getCharacters() { /* ... */ }
  public int getLines() { /* ... */ }
  public Triple<Integer, Integer, Integer> getAll() { /* ... */ }

};
// implicitly, deallocation is done by unreferencing + GC
```

Figure 1: Two stylistic variants of the same interface

```
word_counter_new("README") = 0x9cd6180[struct wc]



word_counter_get_words(0x9cd6180[struct wc]) = 311
word_counter_get_characters(0x9cd6180[struct wc]) = 2275
word_counter_get_lines(0x9cd6180[struct wc]) = 59
word_counter_get_all(0x9cd6180, 0xbffeed00[stack],
   0xbffeecfc[stack], 0xbffeecf8[stack]) = 0
word_counter_free(0x9cd6180[struct wc]) = ()
```

```
_Jv_InitClass(..., 0x6015e0[java::lang::Class], ...) = ...
_Jv_AllocObjectNoFinalizer(..., 0x6015e0, ...) = 0x9158d20
WordCounter::WordCounter(java::lang::String*)(
   0x9158d20[WordCounter], 0x9ae3dc8[java::lang::String]) = ()
WordCounter::getWords()(0x9158d20[WordCounter]) = 311
WordCounter::getCharacters()(0x9158d20[WordCounter]) = 2275
WordCounter::getLines()(0x9158d20[WordCounter]) = 59
WordCounter::getAll()(0x9158d20[WordCounter]) = 0x9f6093e8[Triple]
```

Figure 2: Traces generated by a simple client of each interface

identifiers differ. Moreover, the components are written in different languages, so compilation will introduce further differences. Calls to the Java component will use virtual dispatch and exception handling, while C code will not.

These conventions are not invented anew by each programmer. Rather, they are imported from a cultural repertoire, defined by a language, a toolchain, or simply a coding style. We want to capture each convention in a one-time effort, so that programmers need consider only an abstracted, style-independent view during composition tasks. We can consider this abstraction as a rewriting exercise on *traces* of the kind shown in Fig. 2, which are an annotated extension of the traces generated by the well-known ltrace tool[1]. Although stylistic variation is a broad phenomenon, this trace view captures a large subset of it.[2]

In more realistic examples, there will be not only stylistic differences, but also differences in how each programmer has modelled the domain. These are precisely what is handled by style-unaware adaptation tools [7, 15–17, 21, 23]. Style support complements such tools; Fig. 3 illustrates this. Styles may be captured as "views" or "lenses" which abstract "vertically", recovering a more abstract interface from a more concrete one. Horizontal adaptation can then be performed as usual, *but at the more abstract level.*

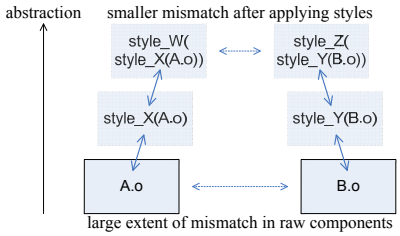

Figure 3: Styles as abstractions over interfaces

[1] http://www.ltrace.org/

[2] The use of pointers in the traces is an abbreviation; the trace properly includes the full exchanged data structures.

Abstract concern	Sample concretion approaches	Concrete examples
error output	thrown exception	exceptions in C++, Java, ...
	use subspace of return value	return null pointer, negative integer, etc
input parameters	immediate	parameters on stack (most language impl'ns)
	indirected	pass by reference (C++, ...)
output parameters	single return value	direct register- or stack-based return (C, ...)
	return packed structure (tuple, array, etc.)	tuple return
	write to locations passed as input	(most C APIs' output parameters)
name word-separating	punctuation-based	underscores (most C APIs)
	casing-based	camel casing
call dispatch (demux)	direct call to single target (trivial dispatch)	(any non-virtual, non-overloaded, singleton functions)
	unary look-up in object	object points to table of target functions (e.g. vtable)
string data type	(generic sequence, inst'd for characters)	Haskell strings; C strings (delimited sequence of chars)
	distinguished abstract data type or builtin	strings native to Java, Pascal, many other languages
data sequence termination	stored sequence length	length field in Java arrays; C++ arrays impl'd with cookie
	special delimiter value	null-terminated character strings in C
object initialization	ad-hoc functional abstraction, integrated with memory acquisition	GObject-style constructors (<classname>_new())
	systematic functional abstraction	C++ constructors, Java constructors
memory acquisition	system functional abstraction or builtin	C malloc(), C++/Java new, ...
	ad-hoc functional abstraction per ADT	GObject-style constructors (<classname>_new()) (other in-ADT allocation functions)
memory release	explicit free using per-ADT functional abstraction	(<classname>_delete() functions supplied by many APIs)
	implicit free by unreference (+GC)	garbage collection in Java, Lisp, ML, ...

Table 1: Stylistic concerns relevant to Fig. 1

Any interface convention which recurs across a large population of components may be considered a style. What interface conventions recur in this way? This question can only be answered empirically. There are no prior studies on stylistic variation. The Appendix presents a preliminary catalogue of stylistic concerns gathered from simple programming experience. For each *concrete* convention we observe, we can identify an *abstract* concern that it models. Note that our catalogue need not be exhaustive. Our approach captures *user-defined* styles—using the list as a guide, but not limited to it. To give a flavour, Table 1 shows a slice of this table containing the conventions evidenced in Fig. 1.

3. APPROACH

Our approach consists of four parts.

A unifying medium which could be any intermediate or bytecode-like representation of code. Relocatable object code, augmented with debugging information, is the one we adopt. This is output by many implementations of a wide range of languages. (Note that our black-box approach works purely by link-time insertion of generated code, and is architecture-agnostic.)

A language for describing styles which we develop as an extension to the Cake composition language [16]. Cake code consists of rules which relate one component interface to another, by identifying *corresponding* data types and function calls. Cake rules conceptually specify a *transducer* which rewrites *traces* like those in Fig. 2. Adding support for styles means extending Cake to *multi-hop* relations, formed by multiple transducers. Rather than relating one fixed interface to another, style rules relate elements of a *more concrete* interface to a *more abstract* one, and are parameterised so they can apply to any component modeling a style.

A language for describing compositions in terms of the styles they instantiate: our composition language is again based on Cake. The programmer introduces a component with an exists declaration, as in normal Cake code, but now including an ordered list of named styles which the component models. The order is used to determine coarse-grained precedence. Our semantics handles the fine-grained composition of styles.

Semantics for the combination of these: given some style definitions and a composition annotated with the styles of each component, the composition formed by our tool is defined by an *elaboration* process. Informally, this is a backtracking search for the "most abstracting" path by which a given function call or data value could be transmitted between the composed modules, given the styles that the programmer has applied (and any horizontal rules that have been defined). We will briefly illustrate this process by example in the next section.

4. EXAMPLES

First, we consider a simple data representation concern, and second, more complex styles concerning function calls.

4.1 Booleans

A simple example of styles concerns encoding of booleans. For example, C code often encodes booleans as integers, with zero indicating false and nonzero indicating true. An opposite convention exists in Unix shell programming: zero indicates truth, and nonzero indicates falsehood. Fig. 4 shows two style definitions capturing these two alternative conventions.

The styles use Cake's table construct to relate enumerated sets of values. This is the relational analogue of an enumerated type: rather than enumerating a set of possible values, it enumerates *correspondences* between elements of one data type and those of another. Style rules relate two *views* of the same component: a more concrete view (always on the left) and a more abstract (on the right). Styles may be parameterised (in a macro-like fashion) to widen their applicability,

```
style c89_booleans(integer_typename)
{  table integer_typename ⟷ boolean
   {   0 ⟷ false;
       _  ⟶ true; /* ordered pattern-matching */
       1 ⟵ true;
}; };

style shell_booleans (integer_typename)
{  table integer_typename ⟷ boolean
   {   0 ⟷ true;
       _  ⟶ false;
       1 ⟵ false;
}; };

exists //↙ apply c89 style, parameter "BOOL", to...
  c89_booleans(BOOL)( //↙ ... the underlying component
            elf_reloc ("componentA.o")
  ) componentA; // ⟵ identifier for the ensemble

exists //↙ apply shell style, parameter "BOOL", to...
   shell_booleans (BOOL)( //↙ the underlying component
            elf_reloc ("componentB.o")
  ) componentB; // ⟵ identifier for the ensemble

derive my_composition = link[componentA, componentB]
{ /* "horizontal" composition- specific rules go here */ };
```

Figure 4: Two styles, and their use in a composition

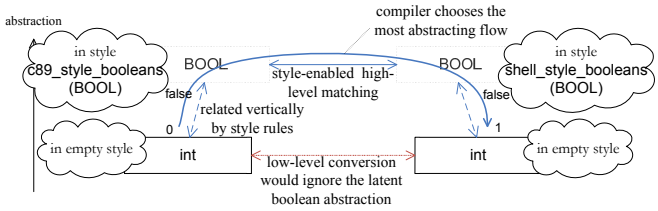

Figure 5: Elaboration of the most abstracting flow

and these parameters are supplied at exists-time. Our styles are parameterised on an identifier (integer_typename) used to identify the data type that is encoding booleans as integers.[3] The exists and derive declarations introduce two components, componentA and componentB, each representing booleans as integers, but where componentA uses the C conventions, whereas componentB uses the shell conventions. Mismatch is avoided by applying the appropriate styles, allowing the Cake compiler to generate conversion logic.

How does the compiler work out what rules should apply to these integers? This is determined by the *elaboration* of styles. In our example we have two possible "flows" for a BOOL: one treating it in the style-specified way, and one treating it as a plain integer. Fig. 5 illustrates. When compiling this, the compiler must select a particular *sequence* of value conversion rules. For each component, it chooses from

[3]This is not simply int for two reasons. Firstly, languages other than C name integers differently. Secondly, not *all* integers are really booleans. For now we are assuming that some quasi-annotation has been done for us, e.g. by a C programmer using typedef to create a synonym for integers, namely BOOL, used exactly when they represent booleans. For other cases, the Cake language has features for annotating distinguished use contexts of a given data type, which we do not discuss here.

```
1  style  jni_static_long_call  (classname, funname, argsig)
2  { // ↙ guard predicates        names bound to return values ↘          ↙ patterns on contextual calls
3   [status != JNI_ERR]                              (status, jvm, env) ⇐ JNI_CreateJavaVM(_, _, _),  ...,
4   [c    != 0, @FindClass == (*env)↪FindClass]                    c ⇐ @FindClass(env, #classname),  ...,
5   [mid != 0, @GSMID == (*env)↪GetStaticMethodID]               mid ⇐ @GSMID(env, c, #funname, #argsig), ...,
6   [@CSLM == (*env)↪CallStaticLongMethod]                          @CSLM(env, c, mid, args... ) // the "triggering" call
7                                          ⟶  classname ## _ ## funname ## _ ## argsig(args...);
8                                          //      ↖ the abstracted view: a single call, named by cpp-style metaprog'ing
9    // extra rule needed to allow reversibility
10   JNI_CreateJavaVM(out _, out _, my_vmargs) ⟶ {};
11 };
```

Figure 6: Abstracting a sequence of calls

```
JavaVM *jvm; JNIEnv *env;
JavaVMInitArgs vmargs;
long st = JNI_CreateJavaVM(&jvm, &env, &vmargs);
if (st != JNI_ERR)
{  jclass  c = (*env)->FindClass(env, "java/lang/System");
   if (c)
   { jmethodID mid = (*env)->GetStaticMethodID(
        env, c, "currentTimeMillis", "(J)V");
     if (mid)
     {    jint  result = (*env)->CallStaticLongMethod(
            env, c, mid, 5);
} } } // else handle errors
```

Figure 7: JNI code for a simple function call

the rules defined by each style applied to that component. Loosely, elaboration searches for a successful composition (i.e. each function call yields a correspondent in the opposing component, and similarly for all data types used) while always preferring a *more abstract* flow. This means preferring a "taller stack" of styles. The order in which the styles were applied is respected. (This logic is near-trivial in our example, since only one style is applied on each side.)

4.2 Java Native Interface style

As a more advanced example of styles, interpreting function calls, consider a caller written in C but consuming a Java library using the Java Native Interface [19]. Fig. 7 shows C code a JNI programmer might write, and Fig. 6 shows a style definition for abstracting the resulting trace into a single call obeying a simple naming convention.

The rule consists of a comma-separated list of patterns, each of which matches a function call and binds names to its elements, including (to the left of the ⇐) its return values. Each pattern is preceded by a square-bracketed *guard predicate* defining additional matching conditions in terms of the names bound in the pattern. Data-dependent matching, i.e. matching only particular argument values, is threaded through the list of patterns by re-using identifiers bound earlier. The final element of the pattern defines the call which "triggers" the rule, here @CSLM[4]; the rule "fires" when this call occurs in a context where calls matching the previous patterns have preceded it. The pattern-list is followed by a right-arrow; on the right of the arrow is the "abstracted" view (line 7) of the left side. Here this is a single call whose name is built from the style's arguments, using metaprogramming operators like those in the C preprocessor.

By applying these rules, we recover an abstract sequence of calls, discarding JNI details. Now we consider the reverse direction—given some abstract sequence of calls, generated by some heterogeneous client in another style (such as a different foreign function interface than JNI), how can we dispatch this against the JNI interface? To avoid introducing another example style, let us simply turn the tables: how do we dispatch abstract calls to JNI? This means running our JNI style rules "in reverse".

In short, we direct the abstract calls into generated stub code whose role is to reproduce a context satisfying the predicates on the left of the JNI rule (lines 3–6). To do so, it keeps a "sliding window"-style log of call history across the interface. For example, on receiving the first abstract call, JNI_CreateJVM has yet to be called, so our stub does this and checks the return value against JNI_ERR. Continuing, we can use the data dependencies between patterns to synthesise the arguments to subsequent calls, using the contents of the call history (which includes earlier argument values). In a few cases, the relevant arguments cannot be recovered without extra programmer guidance; for example, we cannot recover the vmargs argument to JNI_CreateJavaVM. An extra rule (line 10) handles this: the empty right-hand side signifies it may be inserted whenever necessary, and crucially, it provides the required argument value for vmargs, namely my_vmargs. (Here this is assumed to name a statically defined structure in the instantiating component; more realistically, this identifier would be a parameter of the style.)

5. DISCUSSION AND FUTURE WORK

Currently we have only syntax and some paper semantics for styles. However, work on implementing these within the Cake compiler is ongoing. (In fact, styles were an envisaged feature from the initial design of Cake.)

Deeper experience with styles, by further case study, is required in order to discover how our preliminary results generalise. An empirical study of styles found in a large set of codebases (e.g. open-source code in a variety of languages) would be both valuable and feasible.

Performance of generated code is limited by how well the multi-layered glue code generated by our design can be collapsed to a small and efficient adapter, using whole-program optimisation techniques; this requires further research.

What we have loosely claimed to be a "style" is really describing a "style transformer": a mapping from one style to another, where the latter is hopefully more abstract. For

[4]Here identifiers beginning with "@" are treated as metavariables, rather than resolving to component-level names; line 5 binds @CLSM to env's GetStaticMethodID member.

example, the naming convention we selected in Fig. 6 is itself another style, even though it has discarded JNI details. It is therefore essential that styles compose with each other, that *mismatch between styles* does not become a problem, and that a quadratic explosion of styles can be avoided. The emergence of "well-known" *named styles*, into which a wide stylistic variety of input components can be transformed, might solve this analogously to how popular intermediate file formats can avoid quadratic explosion in Make [12].

It would be useful to automatically infer what known styles apply to a component, by searching for the relevant patterns in interfaces. This search becomes more complex when considering compositions of styles and parameterisation. A likely solution might combine backtracking search (much like Make finds compositions of rules satisfying prerequisites) with constraint solving (to find satisfying instantiations of styles' parameters). Similarly, it would be useful to automatically infer likely styles, given a corpus of interfaces, perhaps using existing learning approaches [11].

Assurances about style-based compositions could be gained by considering their round-tripping properties, as with *lenses* for tree-structured data [13]. One idea is to *cross-check* round-trips using symbolic execution techniques [8].

6. RELATED WORK

Component systems such as CORBA [22] use stub compilers to abstract interfaces, but fundamentally do not address heterogeneity, since they assume all components are programmed against interfaces *generated by* such a compiler. By contrast, styles both generate abstractions and *recognise concretions*, enabling heterogeneous composition.

Kent identified a similar phenomenon to stylistic variation in database schemas [18]; we have effectively extended consideration of this phenomenon to component interfaces.

Flexible Packaging [10] has similar goals to ours, but relies on a clean-slate approach to development, whereas our approach is designed to apply to existing components.

The abstracting, normalising nature of styles is similar to the "objectification" transformation of COMPOST [2], but with considerably greater flexibility—notably a language-agnostic, black-box approach.

Interface styles lie on the same spectrum as design patterns [14] and architectural styles [20], but are generally smaller-scale than both. Their small size makes it tractable to describe them in a one-time fashion, but also means that any real interface will feature a complex composition of styles, making style composition a more significant problem.

Composition languages such as Piccola [1] consider how to capture different styles of composition, hence overlapping with interface styles. However, Piccola does not facilitate heterogeneous composition; rather, it formalises compositions within a *single* "compositional style" at a time.

LayOM [5] shares some conceptual similarities, but different objectives: since it does not address heterogeneity, it does not adopt a unifying medium, does not prioritise the definition of new layers (doing which entails C++ source code transformation), and has no analogue of elaboration for automatic composition across layers.

7. CONCLUSIONS

Styles are a novel way to abstract away recurring differences in diverse component interfaces. Our next step is to implement and practically evaluate styles. A survey of observed styles in existing code will add focus to this work. We believe styles can open up a hugely bigger space of feasible compositions than allowed by current tools.

Acknowledgments

I thank David Greaves and Derek Murray for helpful discussion, Aistis Simaitis for feedback on presentation, and the Oxford Martin School Institute for the Future of Computing for support in preparing this manuscript.

References

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

[2] U. Assmann, T. Genssler, and H. Bar. Meta-programming grey-box connectors. In *Proc. 33rd Int. Conf. on Technology of Object-Oriented Languages (TOOLS 33)*, pages 300–311, 2000.

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

[4] M. Blume. No-Longer-Foreign: Teaching an ML compiler to speak C. *ENTCS*, 59(1):36–52, 2001.

[5] J. Bosch. Superimposition: a component adaptation technique. *Inf. and Softw. Tech.*, 41:257–273, 1999.

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

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

[8] C. Cadar, D. Dunbar, and D. Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. 8th OSDI*, pages 209–224. USENIX Association, 2008.

[9] J. Callahan and J. Purtilo. A packaging system for heterogeneous execution environments. *IEEE Trans. Softw. Eng.*, 17:626–635, 1991.

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

[11] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Softw. Eng.*, 27, 2001.

[12] S. I. Feldman. Make: a program for maintaining computer programs. *Softw: Pract. Exper.*, 9, 1979.

[13] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *Proc. 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 233–246. ACM, 2005.

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

[15] J. Järvi, M. Marcus, and J. Smith. Library composition and adaptation using C++ concepts. In *Proc. 6th Int. Conf. on Generative Programming and Component Engineering*, 2007.

[16] S. Kell. Component adaptation and assembly using interface relations. In *Proc. 25th ACM Conf. on Systems, Programming Languages, Applications: Software for Humanity*, 2010.

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

[18] W. Kent. The many forms of a single fact. In *34th IEEE Computer Society Intl. Conf. Digest of Papers.*, February 1989.

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

[20] D. Perry and A. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Softw. Eng. Not.*, 17, 1992.

[21] J. Purtilo and J. Atlee. Module reuse by interface adaptation. *Softw. Pract. Exper.*, 21:539–556, 1991.

[22] N. Wang, D. C. Schmidt, and C. O'Ryan. Overview of the CORBA Component Model. In G. T. Heineman and W. T. Councill, editors, *Component-based software engineering: putting the pieces together*, pages 557–571. Addison Wesley, 2001.

[23] D. Yellin and R. Strom. Protocol specifications and component adaptors. *ACM Trans. Prog. Lang. and Syst.*, 19:292–333, 1997.

APPENDIX

See the author's web page: http://www.cl.cam.ac.uk/~srk31/.